Escuela
Politécnica
Superior

Diseño de comportamientos robóticos mediante deep learning

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Jasmina Rais Martínez
Tutor:
Fidel Aznar Gregori

Universitat d'Alacant
Universidad de Alicante

Septiembre 2019

Diseño de comportamientos robóticos mediante deep learning

Autor
Jasmina Rais Martínez

Tutor
Fidel Aznar Gregori
Departamento de ciencia de la computación e inteligencia artificial

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Septiembre 2019

Preámbulo

En este proyecto se va a estudiar el diseño de comportamiento robóticos con deep learning con el fin de comprobar la bondad de su uso en tareas robóticas. Para ello, vamos a analizar los comportamientos resultantes de aplicar un método de diseño basado en el comportamiento y un método de diseño automático utilizando estrategias evolutivas.

A lo largo del desarrollo del proyecto se utilizarán tanto robots en entornos reales como en entornos simulados en 2D y 3D.

Este proyecto ha sido financiado por el Ministerio de Ciencia, Innovación y Universidades (Spain), proyecto RTI2018-096219-B-I00. Proyecto co-financiado con fondos FEDER.

Agradecimientos

En primer lugar, me gustaria darle las gracias a mi tutor, Fidel Aznar, por su valiosa ayuda a lo largo del desarrollo de este proyecto.

También me gustaría agradecer a mis amigos y compañeros que me han ofrecido su ayuda cuando la necesitaba.

Por último, quiero realizar un agradecimiento especial a mi familia por el apoyo incondicional que me han proporcionado durante toda mi vida.

Una computadora
puede ser llamada
"inteligente" si logra
engañar a una persona
haciéndole creer que es un
humano

Alan Turing

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1. Introducción

1.1. Resumen del trabajo

En este proyecto se ha evaluado el uso de deep learning en tareas de robótica de enjambre. Para ello, se ha desarrollado una conducta de agregación mediante dos métodos de diseño: diseño basado en el comportamiento y diseño automático. Además, se han evaluado diferentes estrategias evolutivas y diferentes parámetros de la red neuronal para la conducta obtenida mediante el diseño automático. Por último, para comprobar el funcionamiento de la conducta de agregación se han utilizado simuladores de entornos 2D y 3D y los robots reales.

1.2. Motivación

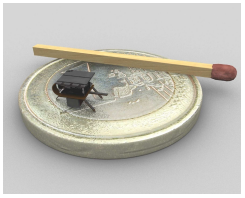
El motivo principal del desarrollo del proyecto es comprobar la bondad del uso de inteligencia artificial en tareas robóticas. Otro de los motivos es poder observar y comparar el resultado obtenido de una tarea diseñada mediante diseño basado en el comportamiento y diseño automático. Por último, el haber elegido la robótica de enjambre para el desarrollo de este proyecto tiene como finalidad observar la comunicación y organización que pueden establecer un conjunto de robots sin el uso de ningún sistema centralizado.

1.3. Estado del arte

La robótica de enjambre presenta diferentes tipos de comportamientos muy variados y donde se pueden utilizar distintos tipos de robots. En este proyecto se ha desarrollado una conducta de agregación mediante dos algoritmos: un algoritmo entrenado con deep learning y un algoritmo basado en el comportamiento de las abejas, Beeclust.

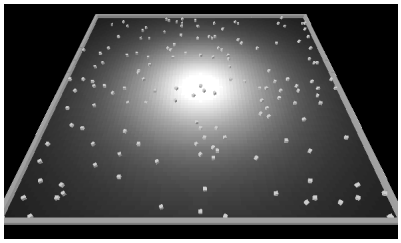
En Schmickl y cols. (2016) se implementó el algoritmo Beeclust utilizando los robots I-SWARM. Este algoritmo reunía a los robots en el área donde la temperatura era la óptima. Además, los robots esperaban un tiempo determinado para alejarse del grupo si no estaba en la localización óptima. El algoritmo se ha utilizado en una simulación de robots Jasmine y en otra simulación con robots I-SWARM. El resultado obtenido en la simulación con robots I-SWARM es el siguiente: Los robots lograron unirse en la localización determinada, de esta forma, se puede comprobar que el algoritmo propuesto en este artículo es robusto y eficiente.

(a) Diseño del robot I-SWARM

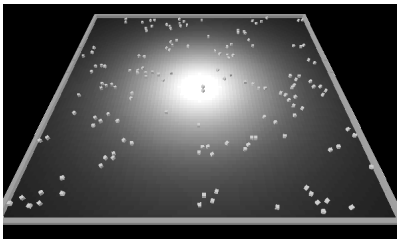


(b) Tamaño del robot en comparación con una cerilla y una moneda

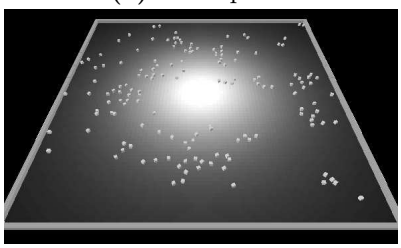
Figura 1.1: Robot I-SWARM utilizado para el algoritmo beeclust



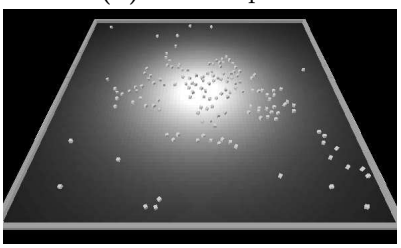
(a) t = 0 pasos



(b) t = 250 pasos



(c) t = 500 pasos



(d) t = 1000 pasos

Figura 1.2: Simulación de 150 robots I-SWARM con el brillo de una lámpara del artículo Schmickl y cols. (2016)

En el artículo SOYSAL y cols. (2007) se utilizó un algoritmo entrenado mediante algoritmos genéticos para llevar a cabo la tarea de agregación y otro algoritmo utilizando métodos probabilísticos. Los sensores que se utilizaron del robot fueron 8 sensores de infrarrojos y 4 micrófonos y los actuadores que se utilizaron fueron ambas ruedas del robot y el altavoz. El comportamiento obtenido con este algoritmo no era escalable porque el grupo el número de robots que se agregaban disminuía al aumentar el tamaño del enjambre.

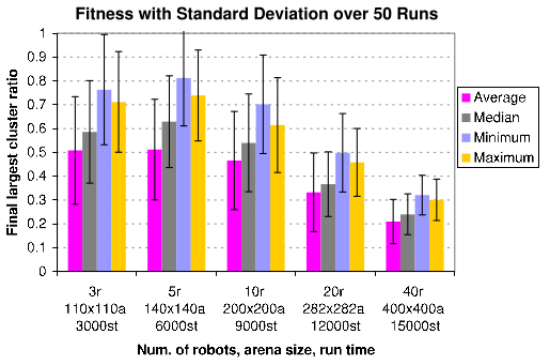
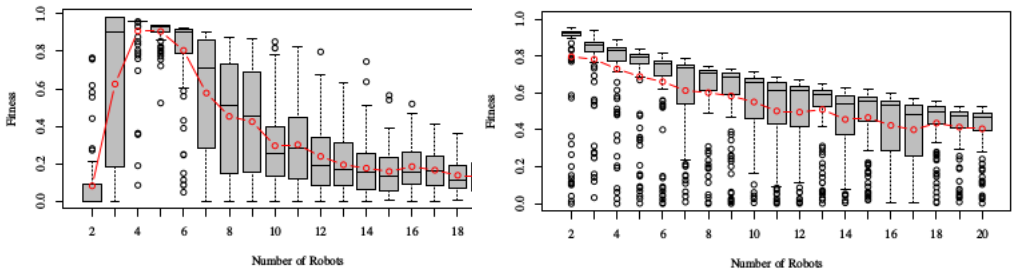


Figura 1.3: Resultados obtenidos en el artículo SOYSAL y cols. (2007)

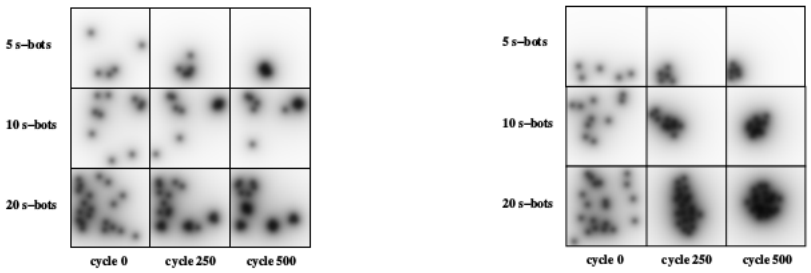
Por último, en Trianni y cols. (2003a) también se utilizaron algoritmos genéticos para el control del robot en la tarea de agregación. El número de robots que se agregaba disminuía al aumentar el tamaño del enjambre, como ocurría en el artículo SOYSAL y cols. (2007). Sin embargo, había diferencias en los resultados entre un cluster dinámico, este es un cluster en el que los robots exploran el entorno juntos una vez se han agregado, y otro estático, en este tipo de clusters los robots se paran en una localización cuando han encontrado al enjambre agregado. El comportamiento obtenido en un cluster dinámico presentaba mejores resultados que el obtenido en un cluster estático.



(a) Recompensas obtenidas en un cluster estático

(b) Recompensas obtenidas en un cluster dinámico

Figura 1.4: Recompensas obtenidas en el artículo Trianni y cols. (2003a)



(a) Comportamiento obtenido en un cluster estático

(b) Comportamiento obtenido en un cluster dinámico

Figura 1.5: Comportamiento obtenido en el artículo Trianni y cols. (2003a)

1.4. Propuesta y objetivos

En este proyecto se propone el aprendizaje y pruebas reales de varias conductas de enjambre mediante Deep Learning y Estrategias Evolutivas para el aprendizaje por refuerzo. Para ello, los objetivos que se establecieron son los siguientes:

- **Diseño del robot con características específicas para los experimentos**: Para cada experimento el robot tiene que tener unas características determinadas, es decir, necesita sensores y actuadores distintos según la tarea que debe desempeñar.
- **Equilibrio individual**: Con el objetivo de probar la plataforma robótica en un entorno inestable se implementará un equilibrio de un robot en un balancín.
- **Equilibrio colectivo**: Para poder observar el comportamiento de varios robots en un entorno inestable necesitamos implementar un equilibrio con más de un robot.
- **Diseño de una conducta de agregación**: Un método de diseño muy utilizado en la robótica de enjambre es el método de diseño basado en el comportamiento, por lo tanto, comprobaremos los resultados que ofrece este método en una conducta de agregación y se comparará este comportamiento con el obtenido mediante aprendizaje automático.
- **Simulación realista**: Para evitar que el comportamiento obtenido en los simuladores sea diferente al obtenido con los robots reales se debe implementar una simulación lo más realista posible. Además, en una simulación realista se puede observar el comportamiento del robot sin poner en riesgo al robot real.
- **Simulación rápida**: Con el fin de acelerar el proceso de aprendizaje se acelerarán los movimientos de los robots en el simulador. De esta forma, obtenemos el mismo comportamiento que con una simulación realista en menor tiempo de entrenamiento.
- **Aprendizaje de una conducta de agregación**: La finalidad de este aprendizaje es encontrar una política óptima para la agregación de un enjambre. Además, se compararán los resultados obtenidos de este algoritmo y los obtenidos por el algoritmo diseñado mediante el método de diseño basado en el comportamiento.

- **Aprendizaje de una conducta de dispersión**: La conducta de dispersión es la contraria a la de agregación, por lo tanto, observaremos el comportamiento obtenido al invertir las recompensas de la tarea de agregación.
- **Diseño de redes neuronales para fomentar el aprendizaje y el funcionamiento del sistema**: Se estudiarán los resultados obtenidos al modificar los diferentes parámetros de una red neuronal con el fin de acelerar el proceso de aprendizaje y encontrar una política óptima.
- **Prueba de varias estrategias evolutivas**: Se probarán varias estrategias evolutivas para poder comprobar cuál es la que mejores resultados proporciona en el menor número de generaciones.
- **Transferencia de una conducta aprendida con simulador simple a simulador complejo**: El simulador simple se utiliza en el proceso de aprendizaje, pero, para poder observar el comportamiento de los robots reales sin ponerlos en riesgo se deberá utilizar el simulador complejo antes de conectar el algoritmo a los robots.
- **Transferencia de una conducta aprendida a robots reales**: Uno de los objetivos de este proyecto es el uso de los robots reales para realizar diferentes tareas, por lo tanto, se debe conectar el algoritmo a los robots después de comprobar que el comportamiento obtenido en el simulador es el correcto.
- **Análisis de métrica de las conductas de enjambre**: La robótica de enjambre tiene varias características como la flexibilidad, escalabilidad y robustez. Por esta razón, se analizarán estos valores en el algoritmo obtenido mediante aprendizaje automático.
- **Pruebas con robots reales**: La finalidad de este proyecto es que los robots logren desempeñar una tarea de enjambre, por lo tanto, se harán pruebas con estos robots para obtener sus características y comprobar que los simuladores se comportan de la misma forma.

Para desarrollar este proyecto se han utilizado conocimientos adquiridos en las asignaturas: Desafíos de la programación, Tecnologías y arquitectura robótica y Sistemas inteligentes.

2. Deep Learning y Deep Reinforcement Learning

En este proyecto se ha utilizado Deep Reinforcement Learning para el diseño de una tarea de robótica de enjambre, es decir, las acciones de los robots se han decidido mediante el uso de una red neuronal. Esta red ha sido entrenada mediante estrategias evolutivas para el aprendizaje por refuerzo.

En este capítulo se va a explicar qué es el Deep learning y los elementos que lo componen, así como el tipo de aprendizaje que se ha utilizado para desarrollar este proyecto y las diferentes estrategias evolutivas utilizadas.

2.1. Introducción

Deep learning es una rama de Machine learning, aprendizaje automático, basado en el uso de redes neuronales artificiales. Tanto el aprendizaje automático como las redes neuronales son un proceso matemático. En este capítulo se explicarán estos conceptos y las técnicas de deep learning utilizadas para el proceso de entrenamiento.

2.2. Redes neuronales

Las redes neuronales son funciones matemáticas inspiradas en las neuronas que constituyen el cerebro humano. En un nivel abstracto se pueden ver como la siguiente función:

$$f_\theta : x \to y \tag{2.1}$$

Donde $\boldsymbol{x}$ es la entrada que produce la salida $\boldsymbol{y}$ y su comportamiento está parametrizado por θ.

2.2.1. Neuronas

Las redes neuronales están formadas por neuronas distribuidas en diferentes capas. La neurona es la parte más básica de una red neuronal y su resultado se puede describir mediante la siguiente fórmula:

$$f\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right) \tag{2.2}$$

Donde $\boldsymbol{x}$ es un vector de entrada que produce un valor escalar y está parametrizado con un vector de pesos, $\boldsymbol{w}$ y un término bias, $\boldsymbol{b}$. Además, la salida depende de la función de activación utilizada.

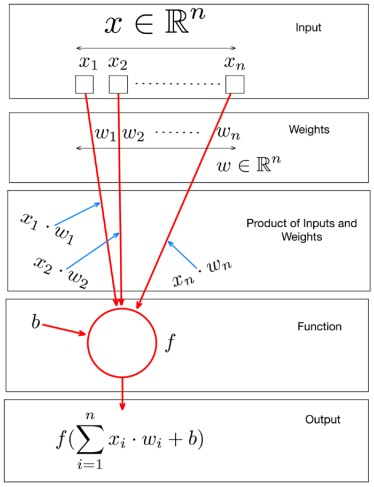


Figura 2.1: Esquema de una neurona

Las neuronas están distribuidas en capas, la última capa es la capa de salida, que está conectada a la última capa intermedia, y la primera capa es la capa de entrada. Además, todas las capas anteriores a la capa de salida deben estar conectadas con su capa anterior y la entrada de cada capa será la salida de la anterior, excepto en la capa de entrada. La anchura de cada capa es el número de neuronas correspondiente y la profundidad es el número de capas.

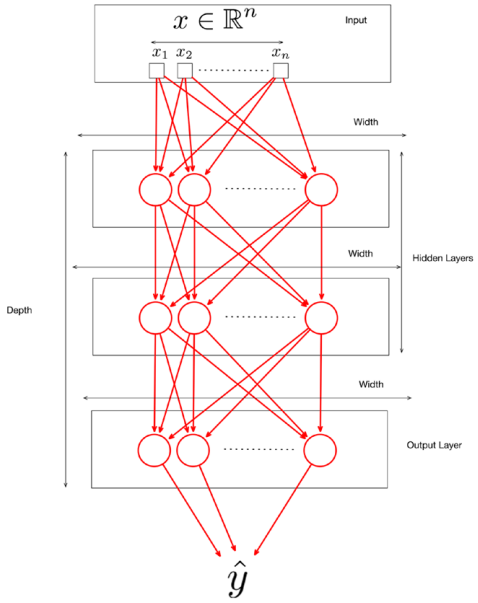


Figura 2.2: Esquema de una red neuronal

Para el entrenamiento de una red neuronal se utiliza el algoritmo de retropropagación para calcular el gradiente de una función objetivo.

2.2.2. Funciones de activación

Las funciones de activación más utilizadas en redes neuronales son las siguientes.

En primer lugar, se pueden utilizar **neuronas lineales**, estas neuronas son las más simples y su función de salida es: $y = w \cdot x + b$. Tienen un comportamiento lineal y permiten que el aprendizaje basado en gradientes sea una tarea más sencilla.

En segundo lugar, se pueden utilizar **neuronas sigmoideas**, su salida modela una distribución Bernoulli sobre $\boldsymbol{y}$ condicionada sobre $\boldsymbol{x}$. Se suele utilizar para problemas de clasificación binaria. El resultado de la salida viene dado por la siguiente función:

$$y = \frac{1}{1 + e^{-(w \cdot x + b)}} \tag{2.3}$$

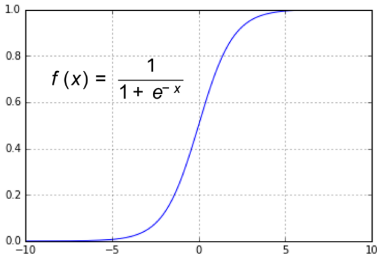


Figura 2.3: Función sigmoid

En tercer lugar, las **neuronas lineales rectificadas** o neuronas ReLU, se utilizan en conjunto con una transformación lineal y se suelen utilizar en las capas intermedias. La salida viene dada por la siguiente función:

$$y = max(0, w \cdot x + b) \tag{2.4}$$

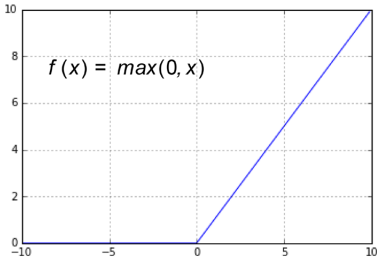


Figura 2.4: Función ReLU

En tercer lugar, la función **tangente hiperbólica**, tanh, es utilizada en las neuronas de las capas intermedias y su función de salida es la siguiente:

$$y = tanh(w \cdot x + b) \tag{2.5}$$

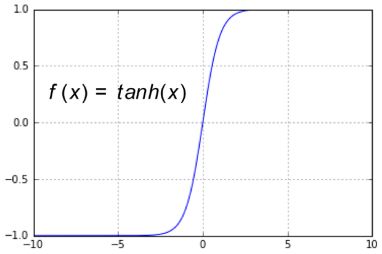


Figura 2.5: Función tanh

Por último, las capas **softmax** son capas utilizadas para la capa de salida de una multi-clasificación, esta capa normaliza las salidas de la capa anterior para que las neuronas en conjunto sumen 1. La capa softmax representa la probabilidad de cada clase.

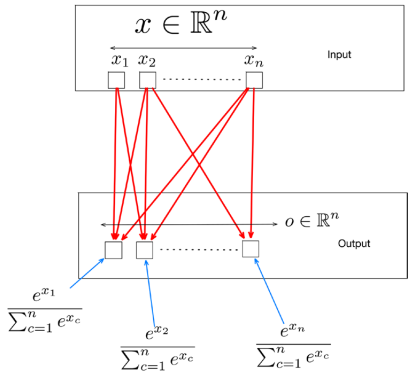


Figura 2.6: Capa softmax

2.3. Aprendizaje por refuerzo

El aprendizaje por refuerzo, RL, es una rama del Machine learning, en el que un agente debe seleccionar las acciones en un entorno con el fin de alcanzar la máxima recompensa global. Este es uno de los tres paradigmas básicos del aprendizaje automático junto al aprendizaje supervisado y al aprendizaje no supervisado. Los algoritmos de aprendizaje por refuerzo buscan maximizar la recompensa obtenida mediante la exploración del entorno.

El proceso de entrenamiento de un agente mediante aprendizaje por refuerzo sigue el siguiente proceso:

1. Inicializar una red neuronal de forma aleatoria.
2. Permitir que el agente interaccione con el entorno.
3. Obtener la recompensa resultante de cada acción generada

De esta forma, obtenemos un registro completo de secuencia de estados, acciones y recompensas. Las políticas que se utilizan en aprendizaje por refuerzo son estocásticas, ya que, solo calculan las probabilidades de ejecutar cualquier acción.

Deep reinforcenment learning, DRL, es una técnica que combina deep learning, o redes neuronales, con aprendizaje por refuerzo con el fin de crear algoritmos eficientes.

2.4. Estrategias evolutivas

Las redes neuronales se pueden utilizar en problemas complejos si se encuentran los parámetros adecuados. Sin embargo, en algunos problemas no se puede entrenar una red neuronal mediante el algoritmo de retropropagación porque no llegaríamos a alcanzar el máximo global y nos quedaríamos estancados en un máximo local. Mediante aprendizaje por refuerzo también se podría entrenar una red neuronal para realizar determinadas acciones que permitan obtener la máxima recompensa, pero, antes se deben calcular los gradientes de las recompensas de cada acción. El problema de está técnica es que, a pesar, de realizar una exploración del entorno también podríamos quedarnos estancados en un máximo local y no alcanzar el máximo global. Como solución, se propone el uso de estrategias evolutivas.

Una estrategia evolutiva es un algoritmo que proporciona una serie de candidatos de posibles soluciones a un problema dado. La evaluación se basa en una función objetivo y proporciona un valor de aptitud. Según el valor de aptitud el algoritmo producirá soluciones candidatas en las siguientes generaciones que será más probable que proporcionen mejores resultados que la generación actual. Este algoritmo sigue un proceso iterativo que finaliza cuando el usuario cree conveniente. Las soluciones candidatas de una estrategia evolutiva se muestrean a partir de una distribución cuyos parámetros están siendo actualizados en cada generación.

A lo largo de este capítulo se explicarán las posibles estrategias evolutivas que se pueden utilizar para el aprendizaje por refuerzo.

2.4.1. SES

Estrategia evolutiva simple (SES), se basa en muestrear un conjunto de soluciones mediante una distribución normal con una media μ y desviación estándar fija σ. Inicializamos μ al origen, en cada generación le asignamos el mejor resultado de la población y en las siguientes generaciones se muestrea alrededor de esta nueva media. En general, este algoritmo funcionará en problemas simples. Debido a su naturaleza voraz, desecha todas las soluciones menos la mejor encontrada, de esta forma puede quedarse atascado en un máximo local en problemas complejos. Una posible solución a este problema sería seleccionar un conjunto de mejores resultados en vez de solo seleccionar el mejor de todos los individuos de la población.

2.4.2. Algoritmos genéticos

Los Algoritmos genéticos simples (GA) siguen un proceso natural que desarrolla la teoría de la supervivencia del más fuerte. Su propósito es optimizar un conjunto de parámetros con el fin de encontrar la mejor solución. Estos algoritmos están formados por una población de redes neuronales y en cada generación se selecciona el conjunto de redes que han proporcionado los mejores resultados. El algoritmo sigue el siguiente proceso iterativo:

Código 2.1: Proceso iterativo de un algoritmo genético

```
pop = init_population()

while True:
    evaluate(pop)
    parents = selection(pop)
    children = breed(parents)
    mutation(children)
    pop = parents + children
```

En el algoritmo anterior tenemos los siguientes pasos:

1. ***init_population***: Se genera la población inicial mediante individuos con parámetros aleatorios.

2. ***evaluate***: Se obtiene la recompensa de cada red neuronal.

3. ***selection***: Se seleccionan las redes neuronales que han obtenido mejores recompensas.

4. ***breed***: Obtenemos una lista de hijos para la siguiente generación que ha heredado los parámetros de sus padres.

5. ***mutation***: Se mutan algunos de los parámetros de las redes neuronales descendientes.

6. ***pop = parents + children***: Se forma la nueva población con los padres seleccionados y los hijos obtenidos, de esta forma se descartan las redes neuronales que no han proporcionado buenos resultados en la generación anterior.

Para llevar a cabo este proceso se deben configurar algunos parámetros como el número de individuos que se conservarán para la siguiente generación o la probabilidad de mutación.

Los algoritmos genéticos consiguen tener una variedad de soluciones óptimas para resolver un problema, pero, el elitismo, es decir, seleccionar las mejores redes neuronales, para evolucionar en las siguientes generaciones puede provocar que las soluciones converjan en un local óptimo.

2.4.3. CMA-ES

Estrategia evolutiva de adaptación de la matriz de covarianza (CMA-ES), es una estrategia que toma los resultados de cada generación y modifica el espacio de búsqueda para la siguiente generación. En cada generación se adapta μ y σ y se calcula la matriz de covarianza completa.

El proceso de adaptación se basa en las N_{best} soluciones y en los parámetros de entrada $\boldsymbol{x}$, $\boldsymbol{g}$ es la generación actual y $\boldsymbol{(g+1)}$ es la siguiente generación.

$$\mu_x^{(g+1)} = \frac{1}{N_{best}} \sum_{i=1}^{N_{best}} x_i \tag{2.6}$$

$$\sigma_x^{2,(g+1)} = \frac{1}{N_{best}} \sum_{i=1}^{N_{best}} (x_i - \mu_x^{(g)})^2 \tag{2.7}$$

Este algoritmo es el más popular para la optimización de gradientes. El único inconveniente es el rendimiento que puede tomar si el número de parámetros del modelo a optimizar es elevado, ya que, el cálculo de la matriz de covarianza es de orden $O(N^2)$, aunque actualmente hay aproximaciones que lo disminuyen a $O(N)$.

2.4.4. PEPG

El objetivo de Gradientes de política de exploración de parámetros (PEPG) es maximizar el resultado de una solución muestreada y si el resultado obtenido es un buen resultado, la recompensa de la población muestreada será mejor. La implementación de esta estrategia que se utiliza en este proyecto está basada en Estrategias evolutivas naturales (NES) pero, incorporando el inverso de la matriz de información de Fisher en la regla de actualización del gradiente. Este algoritmo sigue el mismo esquema básico del resto de estrategias evolutivas, es decir, actualizan el valor de μ y σ en cada generación con el fin de conseguir muestrear un conjunto de soluciones. Debido a que no se actualiza el parámetro de correlación, la eficiencia de este algoritmo es de orden $O(N)$, lo que indica que puede ser más rápido que CMA-ES.

2.4.5. OpenES

OpenAI Estrategia evolutiva (OpenES) se basa en establecer la desviación típica estándar, σ, a un número fijo y solo actualiza en cada generación la media, μ. Además, añade ruido gaussiano a los pesos de la red, $\boldsymbol{w}$ para conseguir que las acciones del agente utilicen una política gaussiana.

2.5. Ventajas de las Estrategias evolutivas frente al aprendizaje por refuerzo

No necesitan el algoritmo de retropropagación, las estrategias evolutivas solo necesitan saber la recompensa final obtenida y no requieren la ejecución del algoritmo de retropropagación.

Permiten **paralelización**. Las estrategias evolutivas solo requieren que las generaciones se comuniquen unos cuantos valores escalares mientras que en el aprendizaje por refuerzo es necesario sincronizar los vectores de parámetros completos.

Las estrategias evolutivas son **más robustas**. Varios hiperparámetros que son difíciles de configurar en el aprendizaje por refuerzo se evitan en las estrategias evolutivas.

Ofrecen una **exploración estructurada**. En algunos algoritmos de aprendizaje por refuerzo, en especial los de gradientes de políticas, se inicializan con políticas aleatorias. Esto puede provocar que manifiesten fluctuaciones aleatorias, es decir, que aparezca mucha diferencia entre los resultados de las generaciones. En cambio, en las estrategias evolutivas esto no ocurre porque usan políticas deterministas y logran una exploración consistente.

Permite la **asignación de recompensas a largo plazo**. Esto es una solución cuando las acciones se suelen repetir en el tiempo y tienen efectos duraderos.

3. Robótica de enjambre

La robótica de enjambre es un método de diseño de comportamientos robóticos en el que los robots trabajan en conjunto con el fin de desempeñar una tarea. Este método está inspirado en el comportamiento auto-organizado de los grupos de individuos en el reino animal. Los robots que pertenecen a un enjambre pueden llevar a cabo una tarea compleja mediante tareas individuales simples. Por lo tanto, en este capítulo se estudiará la robótica de enjambre y las diferentes tareas que puede llevar a cabo.

3.1. Características

La robótica de enjambre tiene las siguientes características:

- Los robots son autónomos.
- Los robots están situados en un entorno y pueden realizar acciones y modificarlo.
- Los sensores y actuadores de los robots son locales.
- Los robots no pueden tener acceso a un control centralizado o a información global.
- Los robots cooperan para llevar a cabo una tarea.

La principal inspiración de la robótica de enjambre proviene del comportamiento observado en aves, abejas, peces, en general a comportamientos del reino animal. El interés de estos animales sociales proviene del hecho de que exhiben una especie de inteligencia de enjambre. En particular, el comportamiento de estos animales parece ser escalable, robusto y flexible.

Diseño robusto es la capacidad de superar la pérdida de individuos. En animales sociales la robustez es promovida por la ausencia de un líder. Por esta razón, los robots no deben seguir órdenes de un sistema centralizado, ya que, la pérdida de este sistema podría causar la inutilidad del enjambre.

Escalabilidad es la capacidad de poder desempeñar una tarea con enjambres de diferentes tamaños. La pérdida o el aumento del enjambre no debería causar ningún cambio drástico en el comportamiento del enjambre. En el reino animal esto se puede llevar a cabo siguiendo una comunicación y percepción sensorial de forma local.

Flexibilidad es la capacidad de hacer frente a un amplio espectro de entornos y tareas. En el reino animal se consigue esta característica debido a la simplicidad de las tareas de cada individuo que pertenece al enjambre.

Los robots que pertenecen a un enjambre deben ser robustos, escalables y flexibles para poder comportarse como un enjambre, es decir, deben seguir la inteligencia de enjambre del reino animal que caracteriza a los animales sociales.

3.2. Métodos de diseño de comportamiento

El diseño es la fase en la que se planea y desarrolla un sistema a partir de requisitos y especificaciones iniciales. En la robótica de enjambre no existe ninguna forma precisa de establecer un comportamiento individual que consiga el comportamiento colectivo deseado. Actualmente hay dos categorías de métodos de diseño de comportamientos: Basados en el comportamiento y método de diseño automático.

El método de diseño **basado en el comportamiento** tiene tres categorías principales: Diseño probabilístico de máquinas de estados finitos (PFSMs), diseño virtual basado en la física y otro tipo de diseño como programación en un medio computacional amorfo. Este último método considera que los individuos tienen capacidad de cálculo y pueden comunicarse con los individuos vecinos. En ocasiones, el método de diseño basado en el comportamiento sigue un proceso de prueba y error, ya que, hay que ajustar ciertos parámetros de los robots para que realicen la tarea indicada. Además, este método se inspira en el comportamiento de los animales sociales, lo cual es una ventaja porque existen modelos matemáticos y un comportamiento particular que definen esos comportamientos. Además, el proceso de diseño de este método suele seguir la estrategia *bottom-up*, a partir del comportamiento individual consiguen diseñar un comportamiento colectivo, aunque se puede utilizar la estrategia *top-down*, a partir del comportamiento colectivo deseado se consigue obtener el comportamiento individual.

Por otra parte, el método de diseño **automático** se utiliza para generar comportamientos sin necesidad de que ningún desarrollador intervenga en el proceso. En este método hay dos categorías: Aprendizaje por refuerzo y robots evolutivos. El uso de estas dos categorías se explicará más adelante.

3.3. Métodos analíticos de comportamiento

El análisis es una fase esencial en el proceso de la ingeniería. En esta fase se comprueba si el comportamiento obtenido es válido o no. El objetivo final es obtener un enjambre de robots reales que se comportan de la forma deseada y con las propiedades deseadas. Las propiedades de los comportamientos colectivos suelen analizarse mediante modelos microscópicos, macroscópicos o mediante el análisis de los robots reales.

Los **modelos microscópicos** tienen en cuenta a cada robot individualmente, la interacción robot-robot y la interacción robot-entorno. En el campo de la robótica de enjambre se han desarrollado varios niveles de abstracción. El más simple de todos es considerar cada robot como un punto de masa, en entornos 2D tienen fuerzas cinemáticas y en entornos 3D también tienen fuerzas dinámicas y los detalles de cada sensor y actuador son modelados. En los modelos microscópicos los robots son simulados y en las simulaciones se utilizan herramientas para verificar que el comportamiento colectivo obtenido es el correcto.

Los **modelos macroscópicos** consideran el comportamiento del enjambre en conjunto y los elementos de los robots individuales no se suelen tener en cuenta para la descripción del sistema en alto nivel. En este modelo se consideran tres categorías: trabajos que recurren a ecuaciones o diferenciales, trabajos donde se utiliza la teoría clásica de control y estabilidad para probar las propiedades del enjambre y otro tipo de enfoques como trabajos modelados que recurren a otro tipo de métodos matemáticos.

El **análisis de robots reales** es una tarea imprescindible, ya que, es inviable simular todos los aspectos del robot. En los entornos reales, los sensores y actuadores pueden percibir ruido de la información exterior, por este motivo, se debe comprobar la robustez de los sistemas de robótica de enjambre observando cuál es su tolerancia al ruido externo. Además, al trabajar con los robots reales podemos diferenciar entre comportamientos colectivos posibles en la práctica y entre comportamientos que son posibles siempre y cuando hagamos suposiciones no realistas.

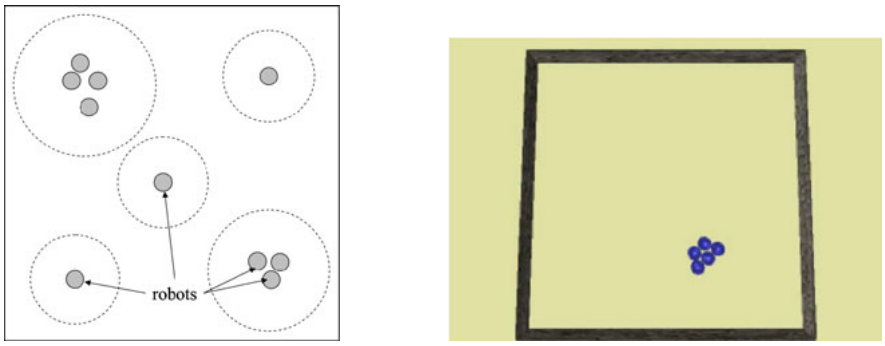
3.4. Comportamientos colectivos

Estos comportamientos colectivos son comportamientos básicos de un enjambre que podrían combinarse con otras tareas más complejas. Estos comportamientos se pueden dividir en cuatro categorías principales: Organización del espacio, navegación, decisión-fabricación colectiva y otro tipo de comportamientos que no pertenecen a ninguna de estas categorías.

3.4.1. Organización del espacio

La **organización del espacio** es un tipo de comportamiento colectivo que se centra en distribuir robots y objetos en un espacio. Los robots se pueden distribuir de distintas formas como agregados, formando cadenas o patrones y estructuras de robots conectados físicamente.

La tarea de **agregación** tiene como objetivo poder agrupar a un enjambre de robots en una región. A pesar de ser un comportamiento colectivo simple, forma un bloque de robots muy útil, ya que, mantiene a los robots muy cerca entre si para que puedan interactuar entre ellos. La agregación es un comportamiento muy común en la naturaleza y suelen realizarlo las bacterias, cucarachas, abejas, peces y pingüinos. En la robótica de enjambre se suele utilizar para el diseño de esta tarea PFSMs o evolución artificial.

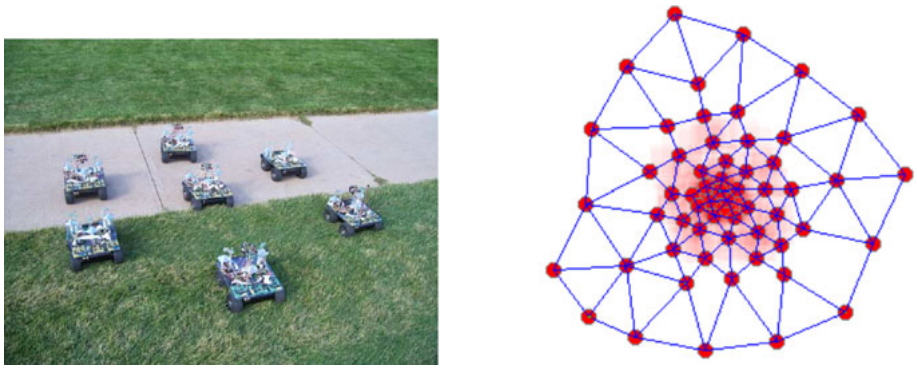


(a) De SOYSAL y cols. (2007) **(b)** De Trianni y cols. (2003b)

Figura 3.1: Ejemplos de la tarea de agregación en robótica de enjambre

La **formación de patrones** tiene como objetivo desplegar robots de manera repetitiva y regular. Los robots necesitan mantener ciertas distancias entre si para formar el modelo deseado. Esta tarea se puede encontrar tanto en la física como en la biología, los ejemplos biológicos son la disposición espacial de las colonias bacterianas y el patrón cromático de golondrinas de mar en el pelaje de algunos animales, mientras que en la física se puede

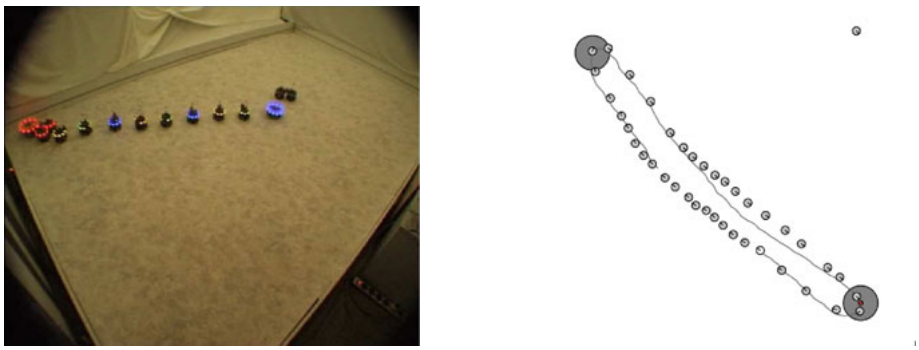
encontrar en la distribución de las moléculas y en la formación de cristales. El método de diseño más utilizado en este comportamiento es el diseño virtual basado en la física. Este método utiliza las fuerzas virtuales para coordinar los movimientos de los robots.



(a) De Spears y Spears (2012) **(b)** De TShucker y Bennett (2007)

Figura 3.2: Ejemplos de formación en patrones en robótica de enjambre

En la **formación en cadena** los robots deben posicionarse en orden para conectar dos puntos. La cadena que forman se puede utilizar como guía para la navegación o la vigilancia. Este comportamiento intenta imitar las cadenas que forman las hormigas. Los métodos de diseño más utilizado en este comportamiento son: PFSMs, diseño virtual basado en la física y evolución artificial. En PFSMs se tienen en cuenta 2 roles: Explorador, que es el primer miembro que aparece en la cadena y guía el movimiento del resto de robots, y miembro de la cadena, mientras que en el diseño virtual basado en físicas se utiliza la distancia entre los robots para modelar el comportamiento.

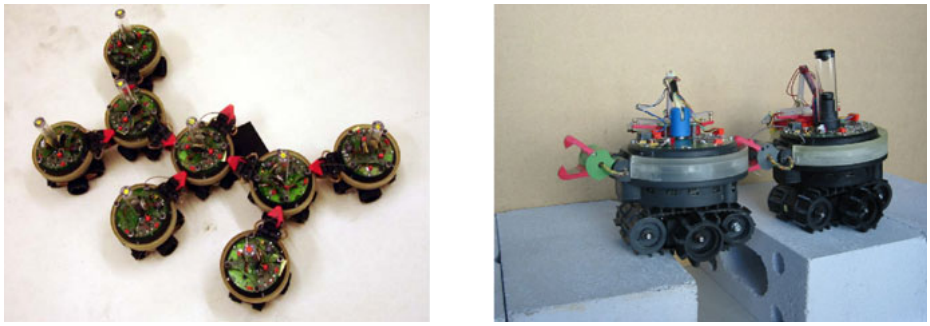


(a) De Nouyan y cols. (2009) **(b)** De Sperati y cols. (2011)

Figura 3.3: Ejemplos de formación en cadena en robótica de enjambre

El **autoensamblaje** es el proceso por el cual los robots se conectan físicamente unos con otros. Este comportamiento tiene diferentes propósitos como aumentar la estabilidad al navegar en terrenos irregulares o para aumentar el poder de atracción entre los robots. La **morfogénesis** es el proceso que lleva a los robots a autoensamblarse después de formar un patrón determinado. Los robots pueden formar una estructura para realizar una determinada

tarea como formar una línea para poder cruzar un puente. El autoensamblaje se inspira en algunas especies de hormigas que son capaces de conectarse físicamente, también se estudia en la auto-organización de las células para formar tejidos y órganos. Desde la perspectiva de la robótica de enjambre hay dos principales desafíos: cómo autoensamblarse en una estructura determinada, es decir, morfogénesis, y cómo controlar la estructura obtenida para abordar tareas específicas. Los trabajos que se centran en el primer desafío se basan en PFSMs y en la comunicación para la coordinación. En cambio, los trabajos que se centran en el segundo desafío utilizan PFSMs o evolución artificial.



(a) De Christensen y cols. (2008) **(b)** De Mondada y cols. (2005)

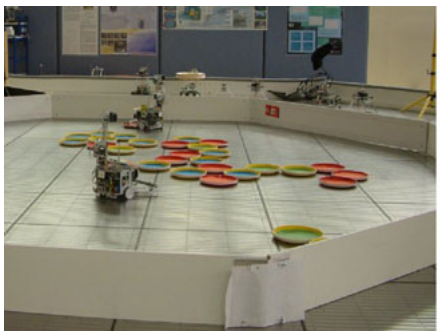
Figura 3.4: Ejemplos de autoensamblaje y morfogénesis en robótica de enjambre

En el **agrupamiento y montaje de objetos** los robots están inicialmente dispersos en un entorno y pueden realizar dos tareas, agrupación o ensamblaje de los objetos, este comportamiento tiene como objetivo agrupar objetos. La diferencia entre agrupar y ensamblar los objetos es que en el primer caso los objetos no están conectados físicamente mientras que en el ensamblaje sí. Este comportamiento es fundamental en cualquier proceso de construcción. Este comportamiento está presente en el reino animal, como en las colonias de hormigas o termitas. En la robótica de enjambre, normalmente se aborda este problema utilizando PFSMs. Los robots exploran el entorno al azar y reaccionan de diferentes maneras al descubrir un objeto o parte de un cluster/ensamblaje de objetos ya formado. En muchos trabajos, esta tarea se realiza de manera secuencial, ya que, el realizarlo de manera paralelizada podría provocar colisiones o interferencias. Para evitar este problema, un robot evita que otros depositen objetos al mismo tiempo que él mediante comunicación o bloqueando físicamente el acceso al cluster.

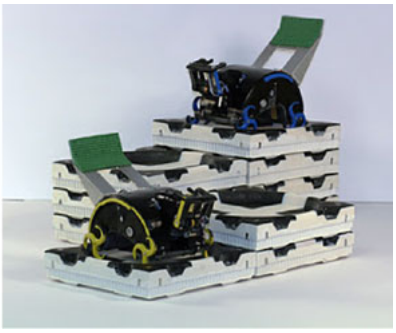
3.4.2. Navegación

La tarea de **navegación** se enfrenta al problema de coordinar los movimientos del enjambre.

La **exploración colectiva** es un comportamiento colectivo en el que los robots cooperan para explorar un entorno mientras navegan. Los comportamientos colectivos que pueden realizar esta tarea son: cobertura del área y navegación guiada por enjambres. El objetivo de cubrir el área es poder crear una cuadrícula regular o irregular de robots para que puedan comunicarse. La cuadrícula resultante se puede utilizar para monitorear el ambiente o para guiar a otros robots. Ambos comportamientos colectivos están fuertemente relacionados, por lo tanto, muchos trabajos se centran en diseñar los dos. La exploración colectiva está presente



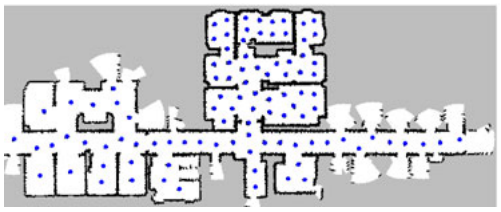
(a) De Melhuish, Welsby, y Edwards (1999)



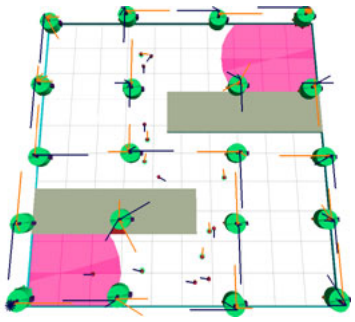
(b) De Werfel (2011)

Figura 3.5: Ejemplos de agrupamiento y montaje de objetos en robótica de enjambre

en el reino animal, por ejemplo, las hormigas usan rastros de feromonas para encontrar la ruta más corta entre dos puntos y las abejas comunican directamente los destinos mediante determinados movimientos. En la robótica de enjambre, es muy común utilizar el método de diseño virtual basado en físicas para poder obtener la cuadrícula de robots que cubre el entorno. Mientras que la navegación guiada se centra en la comunicación, por lo que utiliza PFSMs.



(a) De Howard y cols. (2002)

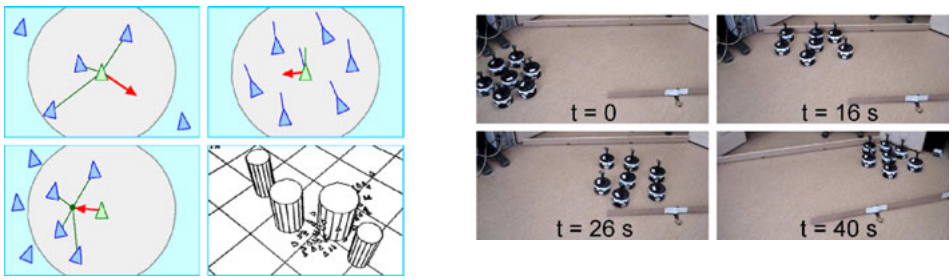


(b) De Ducatelle y cols. (2011)

Figura 3.6: Ejemplos de exploración colectiva en robótica de enjambre

El **movimiento coordinado** se utiliza para que los robots se muevan juntos como las bandadas de pájaros o los bancos de peces. Este comportamiento es muy útil para navegar en un entorno con colisiones limitadas o sin colisiones entre los robots. Además, mejora las habilidades de detección del enjambre. Este comportamiento está inspirado en el observado en las bandadas de pájaros o en los bancos de peces y tiene la ventaja de aumentar la supervivencia del enjambre o aumentar la precisión de la navegación. Este comportamiento suele ser diseñado mediante el método de diseño virtual basado en físicas. Los robots deben mantener una distancia constante entre sí y una alineación uniforme mientras se mueven. Los movimientos coordinados también se pueden obtener mediante evolución artificial.

En el **transporte colectivo** los robots cooperan para transportar un objeto que, en general, es demasiado pesado como para que lo pueda mover un solo robot. Los robots se tienen



(a) De Reynolds (1987) **(b)** De Turgut y cols. (2008)

Figura 3.7: Ejemplos de movimiento coordinado en robótica de enjambre

que poner de acuerdo en una dirección común para mover el objeto hacia un punto objetivo. Se inspira en el comportamiento de las hormigas, cuando estas encuentran un objetivo se ponen de acuerdo para empujarlo a una dirección y si ven que no cambia de posición cambian de dirección a la que empujar. En la robótica de enjambre, se puede diseñar este comportamiento mediante evolución artificial o PFSMs. La cooperación se obtiene mediante comunicación explícita de la dirección deseada o mediante comunicación indirecta, es decir, midiendo la fuerza aplicada al objeto transportado por los otros robots.

(a) De Campo y cols. (2006) **(b)** De Baldassarre (2006)

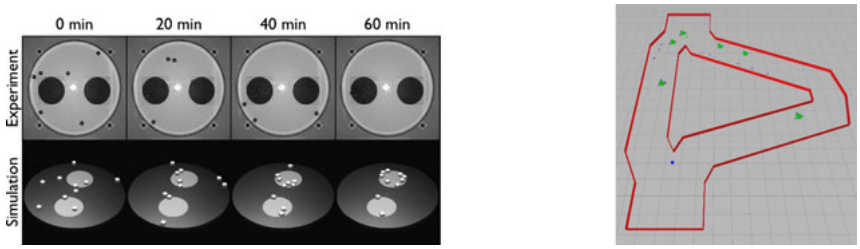
Figura 3.8: Ejemplos de transporte colectivo en robótica de enjambre

3.4.3. Decisión-fabricación colectiva

La tarea de **decisión-fabricación colectiva** trata de como los robots influyen entre si para tomar una decisión. Se puede utilizar para solucionar dos problemas opuestos: acuerdo y especialización. Hay dos tipos de tareas en las que se centra esta tarea: Toma consensuada de decisiones y asignación de tareas.

La **toma consensuada de decisiones** tiene como objetivo que los robots se pongan de acuerdo en tomar una decisión entre varias disponibles. La decisión suele ser la que maximiza el rendimiento del sistema. Esta tarea es generalmente difícil para un enjambre de robots, ya que, la mejor opción puede cambiar en el tiempo o los robots no pueden detectar cuál es debido a que sus capacidades de detección son limitadas. Este comportamiento se encuentra en algunas especies de insectos, como en las hormigas, que utilizan feromonas para decidir

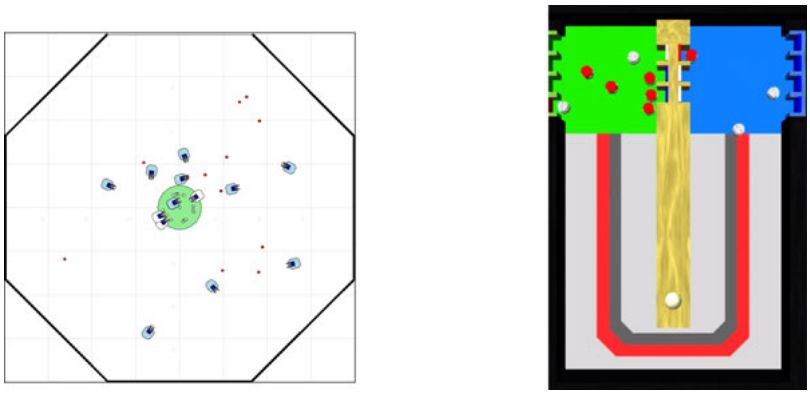
cuál de las dos rutas es más corta, también se puede encontrar en las abejas, que deciden cuál es el mejor alimento o la mejor localización disponible. En la robótica de enjambre hay dos categorías y dependen de la comunicación que se utilice. En la primera categoría se utiliza comunicación directa: cada robot puede comunicar su comunicación preferida o alguna información relacionada. En la segunda categoría se utiliza comunicación indirecta y la decisión se toma a través de ciertas pistas como la densidad de la población de los robots.



(a) De Garnier y cols. (2005) **(b)** De Montes de Oca y cols. (2011)

Figura 3.9: Ejemplos de toma consensuada de decisiones en robótica de enjambre

En la **asignación de tareas** los robots se distribuyen diferentes tareas con el objetivo de maximizar el rendimiento del sistema mediante la elección dinámica de las tareas que desempeñarán los robots. Este comportamiento está presente en las abejas y en las colonias de hormigas, sus decisiones, en principio, son fijas aunque, pueden cambiar en el tiempo. En la robótica de enjambre se suele obtener este comportamiento mediante PFSMs. Para promover la especialización, las probabilidades de seleccionar una de las tareas disponibles son diferentes entre los robots o pueden cambiar en respuesta de ejecución de tareas o mediante mensajes de otros robots.



(a) De Liu (2007) **(b)** De Pini (2011)

Figura 3.10: Ejemplos de asignación de tareas en robótica de enjambre

3.4.4. Otro tipo de comportamientos

Hay otro tipo de comportamientos colectivos que no se pueden clasificar en los mencionados anteriormente. A continuación se describen algunos de ellos.

La **detección colectiva de fallos** tiene como objetivo que el enjambre detecte si uno de los robots presenta fallos o no. El trabajo que ha implementado Christensen y cols. (2009) funciona de la siguiente manera: Todos los robots emiten una señal de manera asíncrona, de esta forma, pueden detectar si un robot está defectuoso recibiendo su señal. Si un robot no está sincronizado, se asume que está defectuoso.

La **regulación del tamaño del grupo** es la capacidad de seleccionar o crear un grupo de un tamaño determinado. Esta tarea puede maximizar el rendimiento para realizar otras tareas. El trabajo que ha implementado Melhuish, Holland, y Hoddell (1999) está inspirado en las luciérnagas. Cada robot emite una señal en un periodo de tiempo aleatorio y después cuentan el número de señales recibidas. Este número puede servir para determinar el tamaño del enjambre.

La **interacción humano-enjambre** tiene como objetivo que un operario humano pueda recibir información del enjambre de robots para conocer el estado del sistema. McLurkin y cols. (2006) desarrolló un mecanismo simple en el que el robot podía proporcionar información mediante el uso de LEDs y sonido.

3.5. Robótica de enjambre con deep learning

Como hemos comentado anteriormente, se puede modelar un comportamiento colectivo mediante deep learnig con métodos automáticos. Las dos técnicas más utilizadas son el aprendizaje por refuerzo y los robots evolutivos.

El **aprendizaje por refuerzo** es un conjunto de algoritmos de aprendizaje. En estos algoritmos un agente explora el entorno mediante una serie de acciones y recibe penalizaciones o recompensas por cada acción, obteniendo al final de cada episodio la puntuación total obtenida con las acciones del agente. En esta estrategia el robot recibe recompensas por sus acciones y el objetivo global es que el robot aprenda una política óptima. El comportamiento es óptimo en el sentido de que maximiza las recompensas recibidas del entorno. Es una tarea difícil poder considerar un problema de robótica de enjambre como un problema de aprendizaje por refuerzo, ya que, nos interesa modelar el comportamiento individual de los robots pero con esta estrategia se establecen las recompensas y penalizaciones en base al comportamiento colectivo. Para solucionar este problema se deben descomponer la recompensa global en recompensas individuales. Aunque esta solución tiene dos problemas: El espacio de búsqueda al que se enfrenta este aprendizaje es grande y esto se debe a la complejidad del hardware de los robots y la complejidad de la interacción robot-robot. El otro problema es que la percepción del entorno es incompleta, esto provoca que la búsqueda del comportamiento sea más compleja.

Los **robots evolutivos**, o evolución artificial, aplican una técnica evolutiva computacional a un solo robot o a un sistema multi-robot. Este método está inspirado en el principio biológico Darwiniano de la selección y evolución natural. Dentro de la robótica de enjambre se han utilizado varias tareas de prueba para probar la efectividad de este método o como una herramienta para contestar preguntas científicas más fundamentales. Este método sigue los

siguientes pasos.

1. Al principio, en la primera generación, se genera una población de individuos cuyos parámetros se han inicializado de forma aleatoria.
2. En cada iteración se ejecuta un número de experimentos por cada individuo.
3. El mismo comportamiento individual del experimento se ejecuta en todos los robots del enjambre.
4. En cada experimento se obtiene una función de fitness o recompensa del comportamiento colectivo como resultado de cada acción individual.
5. Una vez obtenemos las recompensas de todos los individuos de la población se seleccionan los individuos que han obtenido las mejores en la población, se cruzan para obtener individuos descendientes y se mutan.
6. Se vuelve a empezar el proceso desde el paso 2.

El comportamiento individual de los robots se puede representar mediante una máquina de estados finitos o mediante funciones de fuerzas virtuales. Normalmente, los métodos evolutivos se utilizan para encontrar los parámetros de una red neuronal.

Las desventajas del uso de robots evolutivos son las siguientes:

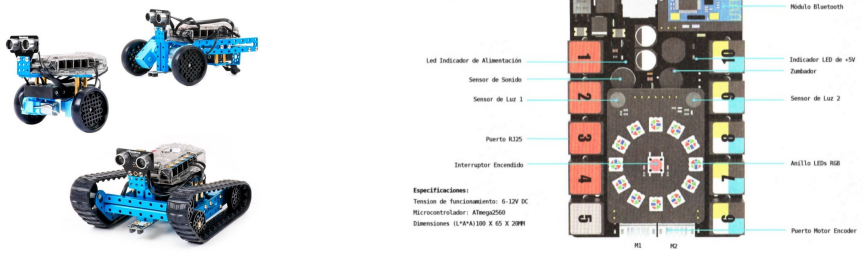
- La evolución es un proceso computacionalmente intensivo y no garantiza la convergencia a una solución.
- Las redes neuronales se comportan como una caja negra (black-box) y a veces es muy complicado entender su comportamiento.
- Desde un punto de vista de la ingeniería, se pueden obtener los mismos resultados diseñando el comportamiento manualmente.

4. Entorno de trabajo

En este capítulo se explicarán los entornos que se han utilizado para desarrollar las pruebas de este proyecto, incluyendo el tipo de robot, la librería de estrategias evolutivas y los simuladores implementados.

4.1. Mbot Ranger

El modelo del robot que se va a utilizar es mBot Ranger, que es un robot diseñado por Makeblock con fines educacionales. Se puede transformar en 3 modelos bases diferentes y ofrece la libertad de poderlo transformar de otra forma completamente diferente. Algunos de los sensores y actuadores están integrados en la placa base Auriga y otros se pueden conectar externamente con módulos electrónicos.



(a) Modelos bases de mBot Ranger **(b)** Placa base Auriga

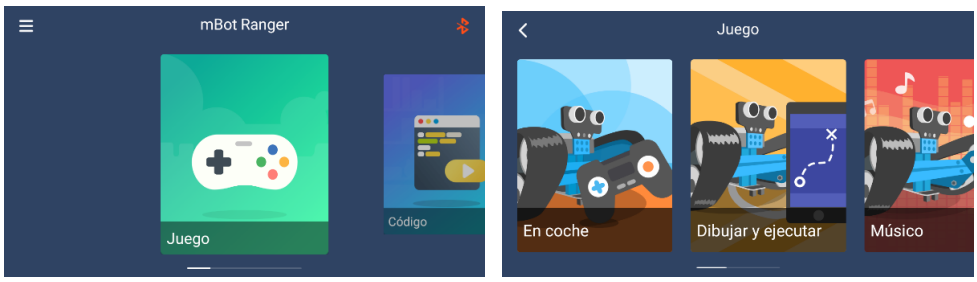
Figura 4.1: Robot mBot Ranger

Se puede controlar mediante 3 formas distintas que se describen a continuación.

En primer lugar, se puede programar mediante la **aplicación móvil MakeBlock**, con esta aplicación se puede controlar el robot de una forma muy limitada, ya que está orientada a que los niños aprendan mientras juegan con el robot. Como este programa controla el robot de una forma muy limitada no vamos a utilizarlo para el desarrollo del trabajo.

En segundo lugar, se puede utilizar el programa **mBlock para ordenador** que está disponible para Linux, MacOs y Windows. Ell único inconveniente es que la última versión disponible para Linux es la v4.0.4, mientras que para los otros dos sistemas operativos es la v5.0.1. Esto es un problema para saber desde que puerto serie está conectado el robot, ya que, en la versión 5 sí que se puede saber desde dónde está conectado, pero en la versión 4 no. Con este programa se pueden programar las instrucciones en Scratch o en Arduino.

En la figura 4.3 se puede observar que en la parte izquierda se encuentra el repertorio de instrucciones de Auriga, en la parte derecha superior se encuentra el código del programa en



(a) Página de inicio de la Aplicación **(b)** Opciones predeterminadas

Figura 4.2: Aplicación para movil

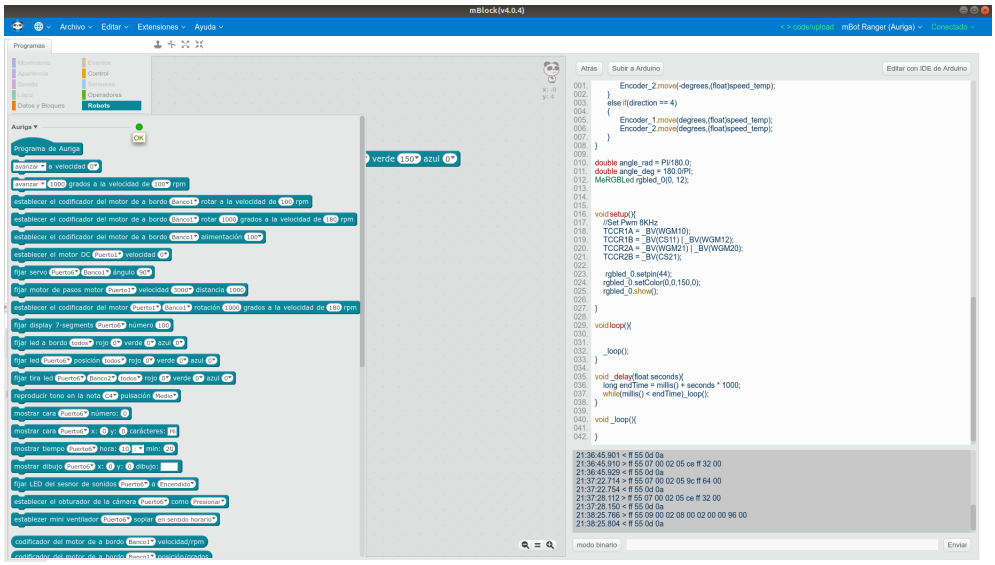


Figura 4.3: Pantalla principal mBlock

Scratch en Arduino y en la parte derecha inferior tenemos la secuencia de envíos de comandos al robot con sus confirmaciones de llegada.

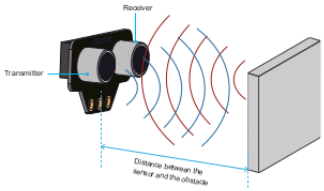
En tercer lugar, se puede utilizar la librería Aurigapy, creada por Fidel Aznar, que traduce los comandos de la placa Auriga a Python. Esta librería es la que se va a utilizar a lo largo del desarrollo del proyecto.

4.1.1. Sensores y actuadores

A continuación se describirán todos los sensores y actuadores del robot Mbot que se han utilizado en este proyecto.

Sensor ultrasónico: Este sensor indica la distancia, en centímetros, a la que se encuentra un obstáculo mediante el envío y la recepción de sondas ultrasónicas. Para calcular la distancia de un objeto primero envía una sonda, después calcula el tiempo que ha tardado en recibir el eco y finalmente utiliza este tiempo para calcular la distancia. Este sensor no se encuentra en la placa Auriga porque es un módulo externo.

(a) Sensor ultrasónico externo



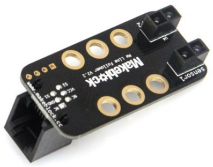
(b) Modo de funcionamiento

Figura 4.4: Sensor ultrasónico

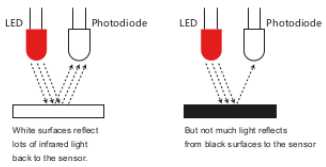
Sensor de luminosidad: Es un foto-transistor cuya señal corresponde a una medición del nivel de luminosidad en el entorno. Sus mediciones son directamente proporcionales al nivel de luz, donde una alta iluminación corresponde a un alto valor de voltaje. Este sensor está incorporado en la placa Auriga.

Figura 4.5: Sensor de luminosidad

Sensor sigue líneas: Este sensor es un emisor-receptor de señales infrarrojas. Su funcionamiento se basa en medir la diferencia de luz reflejada de diferentes colores, solo puede distinguir entre blanco y negro por la configuración digital del módulo Makeblock.



(a) Sensor sigue líneas externo



(b) Modo de funcionamiento

Figura 4.6: Sensor sigue líneas

Giroscopio: Este sensor es un circuito integrado que combina las funciones de un acelerómetro y un giroscopio. Su funcionamiento se basa en medir las proyecciones de la aceleración de gravedad en los ejes del integrado y medir las velocidades angulares mediante un giroscopio. La información que devuelve es el ángulo de giro de los 3 ejes.

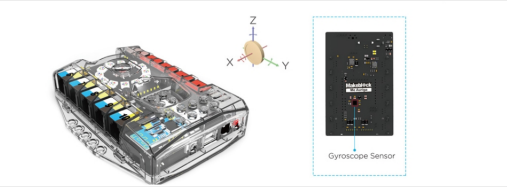


Figura 4.7: Giroscopio

Brújula: Este sensor indica la diferencia angular entre el Norte y la dirección a la que apunta el sensor. Para ello, detecta la intensidad del campo magnético circundante. La información que proporciona es un valor entre 0 y 360º.

Figura 4.8: Sensor brújula externo

Luces LED RGB: La placa dispone de un anillo de 12 luces LED, cada luz es un diodo emisor de luz, que corresponde a un arreglo de 3 diodos de diferente color (R,G,B) cuya suma de colores produce el color final resultante en el LED. Este actuador está integrado en la placa Auriga.

Figura 4.9: Led RGB

Motor con Encoder: El robot dispone de dos motores de corriente continua, que funcionan a través de ser alimentados con un determinado valor de voltaje continuo entre sus terminales y producen un torque proporcional a dicha alimentación. Además incluyen enco-

ders ópticos conectados a sus ejes principales lo cual permite monitorear la posición actual del eje. La dinámica de los encoder se basa en detectar el paso de luz a través de determinadas ranuras, a medida que el rotor va girando, con esta información se pueden calcular velocidades y posiciones absolutas de los ejes de los motores. A los motores se les puede indicar la velocidad de giro en revoluciones por minuto, RPM.

(a) Motor con cable para conexión **(b)** Zoom al eje del motor

Figura 4.10: Encoder Motor

4.1.2. Modificaciones del robot

En este proyecto se ha colocado la placa Auriga en vertical para que los robots puedan comunicarse mediante luz, ya que el sensor de luz está integrado en la placa y no se puede cambiar de sitio de forma individual. La ventaja de esta nueva posición de la placa es que los robots detectan con más intensidad la luz que puede emitir otro robot.

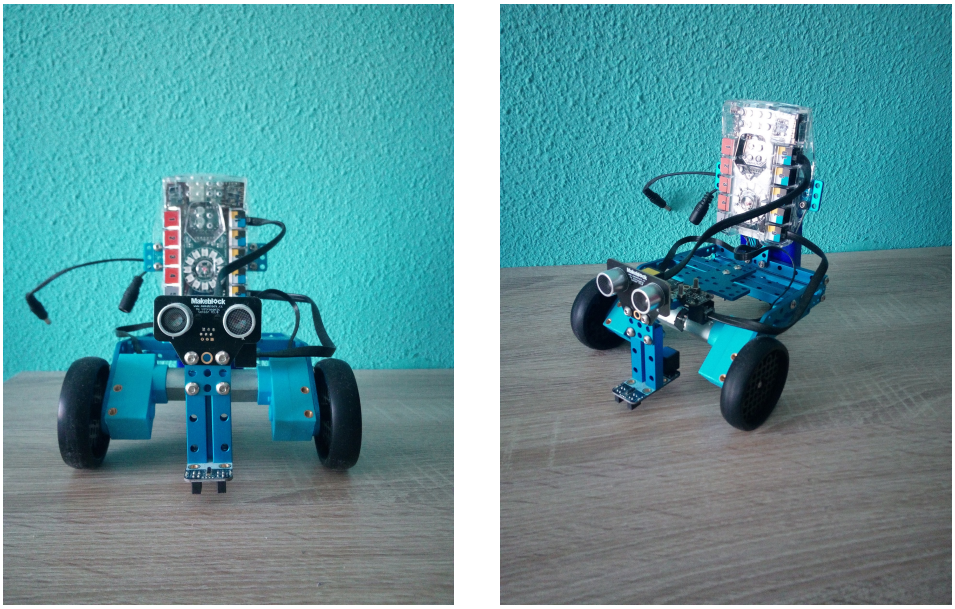
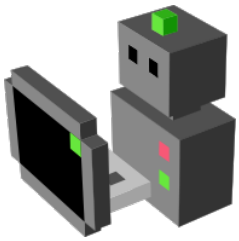


Figura 4.11: Robot mBot Ranger reorganizado

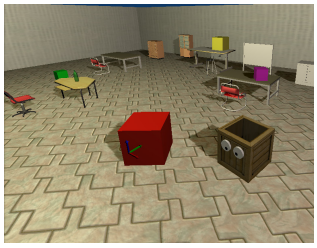
4.2. Simulador MORSE

Para poder realizar pruebas de diferentes comportamientos o entrenar los algoritmos de aprendizaje no se puede utilizar el robot real, ya que, es peligroso porque en ocasiones puede colisionar con algún otro objeto y romperse durante el proceso de entrenamiento. Por lo tanto, la mejor opción es utilizar un simulador para poder utilizar al robot para el proceso de entrenamiento en los algoritmos de aprendizaje.

El simulador que se ha utilizado en este proyecto es MORSE. Este permite el modelado del entorno y de los robots con el programa Blender. Además, permite la comunicación con el robot mediante sockets, de esta forma podemos conocer la información de los sensores o enviar comandos a los actuadores accediendo al puerto que se abre por defecto.



(a) Logo de MORSE



(b) Escenario de una simulación de MORSE

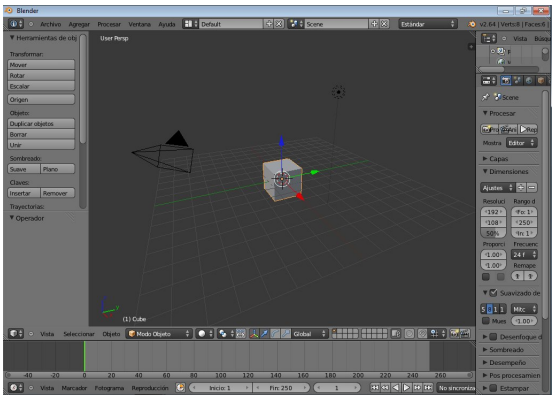
Figura 4.12: Simulador MORSE

4.2.1. Blender

Blender es un programa multiplataforma y gratuito, dedicado especialmente al modelado, renderizado y creación de gráficos en 3 dimensiones. Posee un motor de videojuegos interno y dispone de un sistema de físicas que permite simular fenómenos físicos de entornos reales.

Se ha utilizado este programa para la creación del modelo del robot y de los escenarios.

(a) Logo de Blender



(b) Pantalla inicial de Blender

Figura 4.13: Blender

4.2.2. Modelo del robot

El robot se ha moldeado de forma sencilla y respetando las medidas del robot real para realizar simulaciones lo más realista posible.

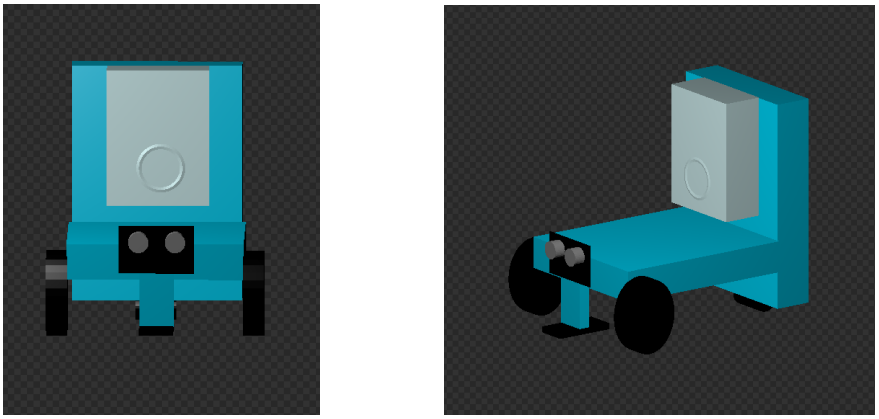


Figura 4.14: Modelo del robot en Blender

4.2.3. Sensores y actuadores implementados

La mayoría de los sensores del robot se han implementado utilizando otros sensores con un script en Python, ya que, no existían en la librería del simulador, los sensores que se han creado son:

Sensor sigue líneas: Este sensor se ha simulado con dos videocámaras colocadas a la misma altura, una a la izquierda y la otra a la derecha. Ambas cámaras pueden capturar un píxel cada una porque la única información que nos interesa es la combinación de colores blanco y negro, ya que, la información que nos proporciona el sensor del robot real es un entero que nos indica que parte del sensor captura pixeles negros o blancos, por lo tanto, sería suficiente con un píxel por cámara.

La información que nos devuelve el sensor implementado es la misma que la del sensor real. El color 0 representa que los bits capturados son de color negro mientras que los bits de color blanco se representan con el color 1.

Cámara izquierda	Cámara derecha	Resultado
0	0	0
0	1	1
1	0	2
1	1	3

Tabla 4.1: Valores del sensor sigue líneas

Como podemos ver en las siguientes imágenes, debido al brillo del suelo no se captura bien el color negro, pero, el color blanco se aprecia perfectamente, por lo tanto, se cuentan los

píxeles blancos y el color negro será el resultado de los píxeles no blancos.

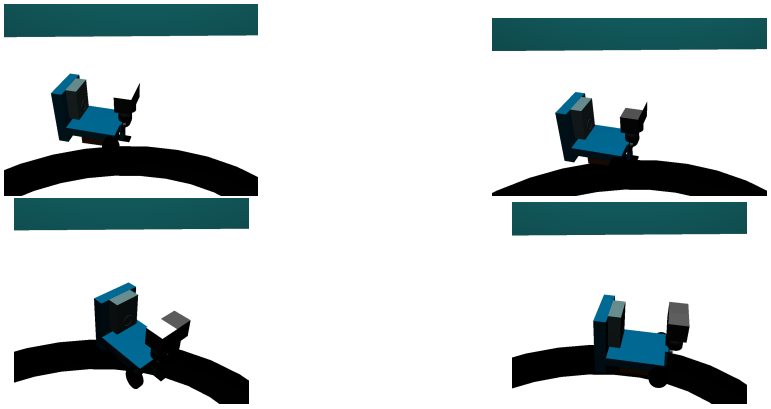


Figura 4.15: Diferentes valores del sensor sigue líneas en MORSE

Las videocámaras se pueden ocultar para no modificar la estética del robot.

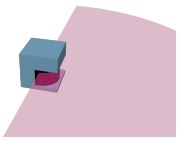
Sensor ultrasónico: En MORSE este sensor no existe y el más parecido es el escáner láser que te indica la distancia a la que se encuentran el resto de objetos. La librería del simulador dispone de 3 tipos de láser que se describirán más adelante. En todos los escáneres láser se pueden modificar los siguientes parámetros:

- **laser_range**: Indica la distancia, en metros, que es capaz de detectar a un obstáculo.
- **resolution**: Indica el ángulo de diferencia entre cada láser del sensor.
- **scan_window**: Indica el ángulo de cobertura total del láser en grados.
- **visible_arc**: Con este parámetro se puede indicar si el láser es visible o no.
- **layers**: Indica el número de capas que puede tener el láser.
- **layer_separation**: Indica la distancia angular de separación entre múltiples capas.
- **layer_offset**: Es la distancia horizontal entre los puntos de escáner de capas consecutivas.

En primer lugar se encuentra el escáner láser **SICK LMS500**, de manera predeterminada puede abarcar una distancia de 30 metros como máximo y dispone de 180 grados de cobertura total y 180 puntos de lectura.

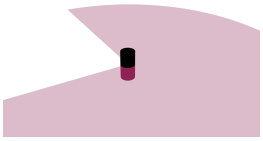
En segundo lugar se encuentra el escáner láser **SICK LD-MRS**, este puede tener varias capas de láseres a diferencia de los otros dos.

En tercer y último lugar se encuentra el escáner láser **Hokuyo**, este dispone de una cobertura de 270 grados y 1080 puntos de lectura. Además, en comparación con los otros dos escáneres, este es más pequeño y sencillo visualmente, lo cual es una ventaja, ya que este componente no se puede ocultar como la videocámara.



(a) Escáner láser SICK LMS500

(b) Escáner láser SICK LD-MRS



(c) Escáner láser Hokuyo

Figura 4.16: Escáneres láser disponibles en MORSE

Finalmente, el escáner láser que va a sustituir al sensor ultrasónico en el simulador es el escáner Hokuyo porque es más sencillo y pequeño que los otros dos escáneres, por lo cual no altera el funcionamiento del robot. Para imitar el comportamiento del sensor ultrasónico necesitamos convertir la distancia del láser a centímetros y solo necesitamos un grado de cobertura y una capa de láseres.

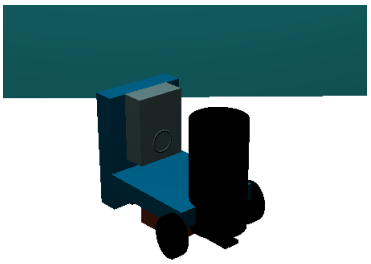


Figura 4.17: Modelo del robot en MORSE con el escáner láser Hokuyo

Sensor de luminosidad: Este sensor indica el nivel de luminosidad que captura en el entorno. El sensor real dispone de dos partes, sensor izquierdo y sensor derecho, cada parte captura un nivel de brillo distinto. Para implementar este sensor se pueden utilizar 6 video-cámaras, 3 para cada parte del sensor, para calcular la intensidad de cada cámara se calcula media de la intensidad capturada, para poder realizar esto último primero se convierte la imagen capturada en blanco y negro y luego se calcula el nivel de brillo de cada píxel para calcular dicha media. Para calcular la luminosidad de cada lado se calcula la media de la intensidad de las 3 cámaras. Cada cámara tiene un tamaño de 16 píxeles de ancho por 16 de alto. Las cámaras apuntan al suelo porque es donde más se refleja la luz emitida por otros robots o por él mismo. El valor de brillo capturado dependerá del entorno y de si tiene las luces LED encendidas o no. Se pueden ocultar las cámaras, como en el sensor sigue líneas, para no modificar la estética del robot.

Figura 4.18: Posición de las cámaras del sensor de luminosidad en MORSE

Brújula: Este sensor indica a cuántos grados de diferencia se encuentra del Norte. Como los anteriores sensores, este tampoco se encontraba en la librería de Morse, pero, sí que se podía sustituir por el sensor GPS que sí se había implementado en la librería. El sensor GPS tiene varias implementaciones y la que incluye la brújula es el nivel Extended. El único inconveniente de este sensor es que no indica el rumbo que sigue el robot si no está en movimiento, por lo que no sabremos a qué dirección apunta si está quieto. Para poder calibrar este sensor hay que indicar la latitud, longitud y altitud del entorno y para evitar que la brújula devuelva un valor infinito cuando el robot está en movimiento se guarda un historial del anterior valor leído. Además, para que el sensor devuelva el valor correcto de la orientación del robot en el entorno simulado, se gira en el eje Z dicho entorno para que el valor del simulador sea el más aproximado al valor real.

Luces LED: Este actuador se puede sustituir con un emisor de luz, que sí está implementado en la librería de Morse. Para que fuera lo más parecido al anillo de LEDs del Mbot Ranger se le asignaron los siguientes valores a los siguientes parámetros.

- **Distance**: Es la distancia a la que se va desvaneciendo la luz, en este caso la distancia es 0.1 metros.
- **Energy**: Es la energía con la que emite la luz. Para simular la luz que emite el anillo de LEDs, le asignamos el valor 5.0 a la energía para que el sensor de luminosidad capture el mismo valor que detectaría en el entorno real.

Inicialmente indicaremos que estén las luces apagadas y se encenderán desde el script de Python.

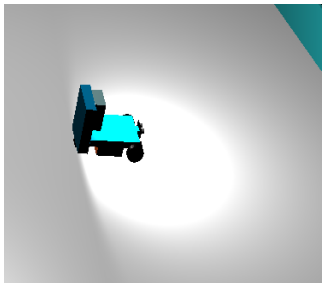


Figura 4.19: Modelo del robot con las luces LED encendidas en MORSE

4.2.4. Escenarios

Se han recreado varios escenarios dependiendo de la tarea que se va a realizar y del número de robots que van a cooperar.

El escenario más simple mide 4 metros de ancho por 4 metros de alto y la iluminación coincide con la de una habitación cerrada sin luz artificial. Este escenario se ha utilizado para comprobar el funcionamiento de algunos de los sensores.

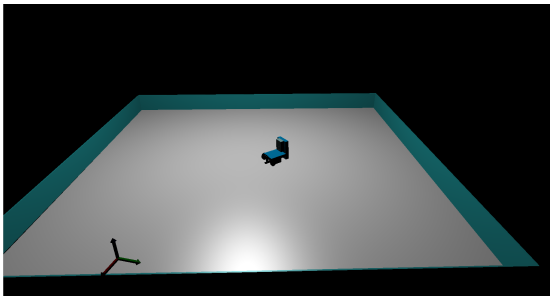


Figura 4.20: Escenario básico en MORSE

Otro escenario recreado es el de un circuito sigue líneas, para que el sensor sigue líneas detecte el color blanco el brillo del suelo debe ser más alto que en el del escenario básico. Además, el mapa sigue líneas es un cilindro hueco, ya que, al intentar poner una imagen del mapa el suelo blanco no se distinguía del negro.

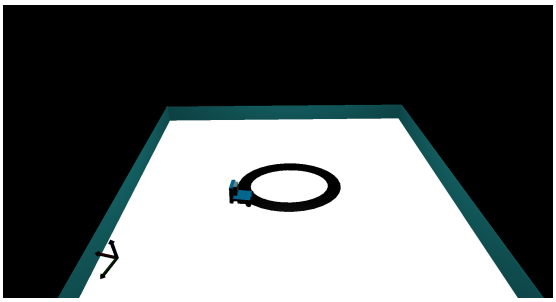


Figura 4.21: Escenario con circuito sigue líneas en MORSE

Por último se han creado varios escenarios para la tarea de agregación, los escenarios tienen el mismo número de robots y la misma iluminación.

Uno de los escenarios que se utilizarán en la tarea de agregación es un escenario cuadrado de 6x6 metros con poca iluminación para que los robots detecten a los demás mediante las luces led. Para esta tarea se han colocado 5 robots en el escenario porque con más robots tardaba mucho más en procesar las peticiones de lectura de los sensores y las peticiones de actualización de los actuadores.

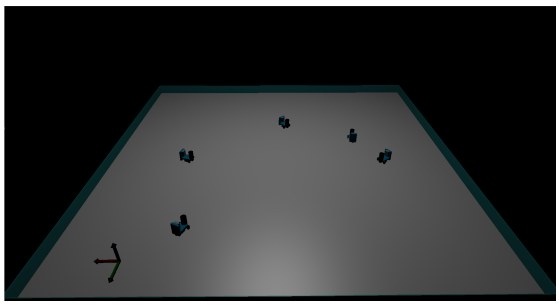


Figura 4.22: Escenario cuadrado para agregación en MORSE

El segundo escenario es una superficie en forma de círculo con un diámetro de 5.6 metros. Aunque el fondo de este escenario no sea negro tiene la misma iluminación que el anterior, ya que el sensor de luminosidad solo comprueba la luz del suelo y el suelo no está más iluminado que en el resto de escenarios.

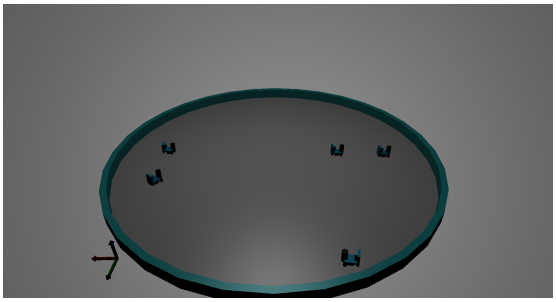


Figura 4.23: Escenario circular para agregación en MORSE

El último escenario es una superficie en forma de cruz con cuatro secciones y de las mismas dimensiones que el primer escenario de la tarea de agregación.

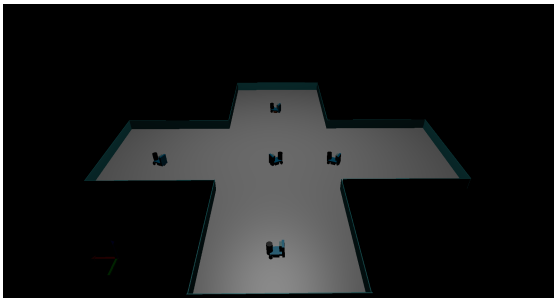


Figura 4.24: Escenario con forma de cruz para agregación en MORSE

4.2.5. Problemas encontrados

Se han encontrado algunos problemas con MORSE a lo largo del desarrollo del proyecto, uno de ellos era el reset del script del simulador y el otro era el bounding box del robot.

4.2.5.1. Reset

Al utilizar este simulador para entrenar un algoritmo genético simple se debía resetear el entorno después de cada ejecución, el problema es que el reset del script generado con Pymorse reseteaba todos los objetos del entorno, incluidos los sensores y los objetos del entorno, por lo que en ocasiones tardaba más de 10 minutos en resetear el robot. Como solución se puede resetear solo la posición del robot, lo cual disminuye considerablemente el tiempo del reset en cada ejecución, ya que tarda menos 1 minuto.

4.2.5.2. Bounding box

Bounding Box es el delimitador del robot, es decir, los objetos colisionan con el bounding box del robot, no con la estructura de este. El problema estaba en que era más grande que el robot y no dejaba avanzar si un obstáculo u objeto estaba a menos de 0.35 metros, lo cual es un problema porque no llega a colisionar y puede perjudicar el proceso de aprendizaje. Como solución se puede desactivar la colisión de *morsy_bb*, es el bouning box que nos genera MORSE y de *morsy* en general y dejar solo activado el de *morsy_mesh*, ya que es la figura que hemos creado.

(a) Distancia máxima con la que podía colisionar el robot antes de modificar el bounding box

(b) Distancia máxima con la que puede colisionar el robot después de modificar el bounding box

Figura 4.25: Distancias máximas permitidas por el bounding box de Blender

4.3. Librería estool

Durante el desarrollo del proyecto se ha utilizado la librería Estool, (Ha, 2017a), para el proceso de aprendizaje del algoritmo de enjambre. Esta librería dispone de la implementación de varias estrategias evolutivas para el entrenamiento de un agente, como se han explicado anteriormente en el Capítulo 2.

Esta librería dispone de varios escenarios implementados y permite añadir otros siempre y cuando permitan el uso de la paralelización y mantengan la misma estructura de los otros escenarios. Los datos, como la mejor red neuronal encontrada o los resultados del proceso de entrenamiento, los guarda automáticamente en un fichero de tipo Json. Además, muestra los resultados obtenidos en cada generación y realiza evaluaciones del modelo cada 25 generaciones para conocer el resultado que obtiene con la mejor red neuronal encontrada.

4.4. Simulador 2D

Como se ha comentado anteriormente, la librería Estool utiliza la paralelización para acelerar el proceso de aprendizaje, por lo tanto, el simulador que se utilice en el entrenamiento debe ser paralelizable para que se pueda utilizar esta librería. Las simulaciones del simulador MORSE no se pueden paralelizar por el uso de los sockets en la comunicación con el robot. Por lo tanto, como la librería es compatible con Box2D, se utilizará un simulador 2D que utilice este motor para poder simular al robot real. Para la creación del simulador 2D se ha utilizado el motor de videojuegos Box2D y la librería PyGame.

Box2D es un motor de videojuegos que permite recrear escenarios y simulaciones en entornos 2D. Esta librería soporta Python y C++ y dispone de un sistema de físicas que permite simular entornos reales. Por otra parte, PyGame es una librería de código abierto que permite crear videojuegos en 2D. Además, permite renderizar los objetos creados con la librería Box2D.

4.4.1. Agente

En este proyecto se ha simulado el robot Mbot Ranger en 2D para poder utilizarlo en el proceso de entrenamiento con la librería Estool. Para ello, se han utilizado las librerías Box2D y PyGame para recrear el agente. En principio, el agente estaba representado como un cuadrado de tamaño 0.5 unidades de ancho y de alto, el problema de esta representación es que al colisionar con una pared no podía girar y cambiar de dirección como hace el robot real y el simulado en MORSE, por lo tanto, para solucionar este problema se cambió la forma del agente a un cilindro de 0.75 unidades de diámetro. Además, se puede renderizar en dos colores, gris y azul, el primero indica que el robot no ha encendido las luces LED y el segundo sí. Para poder almacenar la información del robot, es decir, el estado de las luces LED.

Por último, se han implementado una serie de actuadores y sensores para simular el comportamiento del robot que se explicarán más adelante.

4.4.2. Sensores y actuadores implementados

A continuación, se explican todos los sensores implementados en Box2D.

Sensor de luminosidad: Este sensor indica si ha detectado luz según la distancia del resto de robots y comprobando si alguno de los robots más cercanos ha encendido las luces LED. Para calcular la distancia se utilizan las coordenadas x e y, y se calcula mediante el cálculo de distancias de vectores. La información que devuelve es 0 o 1, es decir, si no detecta luz o si la detecta.

Sensor brújula: Este sensor indica a qué coordenada está orientado el robot. La información que proporciona es un número entero entre 0 y 3, que indican si está orientado hacia

el norte, este, sur u oeste. Para obtener el resultado se obtiene el ángulo en el que está posicionado el objeto en Box2D y se calcula la coordenada correspondiente.

Sensor ultrasónico: Este sensor indica la distancia a la que se encuentra el obstáculo más cercano, para ello, calcula la distancia que hay entre el agente y el resto de objetos de la escena, ya sean robots o paredes. Para obtener la información de este sensor se utiliza de forma auxiliar la brújula, ya que la posición del sensor dependerá de la orientación del robot y dependiendo de la orientación se comprobará la distancia según el eje cartesiano correspondiente. La información que devuelve es un número entero entre 0 y 2, dependiendo de la distancia a la que se encuentre el obstáculo se indicará si está cerca, a distancia media o lejos.

Por otra parte, los actuadores que se han implementado son los siguientes:

Ruedas del robot: Para simular el movimiento del robot se ha utilizado la velocidad angular y lineal. Con estos valores el robot podía desplazarse en varias direcciones sin tener que determinar la velocidad de cada rueda.

Luces LED: En la simulación en 2D, las luces LED son un valor booleano que indica si están encendidas o no. Para distinguir el estado de las luces visualmente se renderizan en distinto color dependiendo de dicho estado.

Para simplificar los datos de entrada y salida de la red neuronal todos los valores se han discretizado. De esta forma, la red puede converger con menos tiempo de entrenamiento, ya que, el espacio de la toma de decisiones se ha reducido.

4.4.3. Escenarios

Se han utilizado diversos escenarios en 2D para el entrenamiento y prueba del comportamiento de agregación.

En primer lugar, se ha utilizado un escenario cuadrado con 5 agentes para la mayor parte del entrenamiento de agregación.

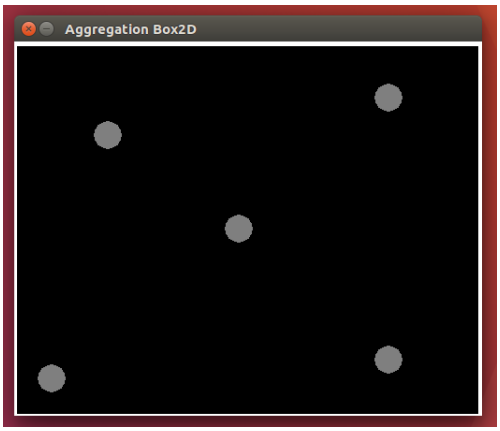


Figura 4.26: Escenario cuadrado para agregación en Box2D

Por otra parte, también se han utilizado otros escenarios para comprobar el funcionamiento del algoritmo de aprendizaje. Uno de estos escenarios es un recinto circular cuyas paredes se han creado con cuadrados de 0.06 x 0.06 dimensiones con 1 grado de separación que formaban un círculo, ya que, en Box2D no existen las paredes circulares. Para determinar las posiciones de cada pared se ha utilizado la siguiente fórmula:

$$\begin{aligned} x_2 &= x_1 + r \cdot sin(\alpha) \\ y_2 &= y_1 - r \cdot (1 - cos(\alpha)) \end{aligned} \tag{4.1}$$

Siendo (x_1, y_1) las coordenadas del punto inicial superior de la circunferencia, $\boldsymbol{r}$ el radio de la circunferencia, α el ángulo de separación entre ambos puntos y (x_2, y_2) la posición donde debe colocarse el siguiente punto.

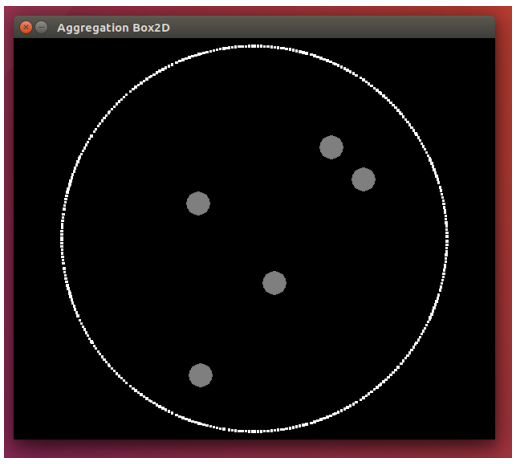
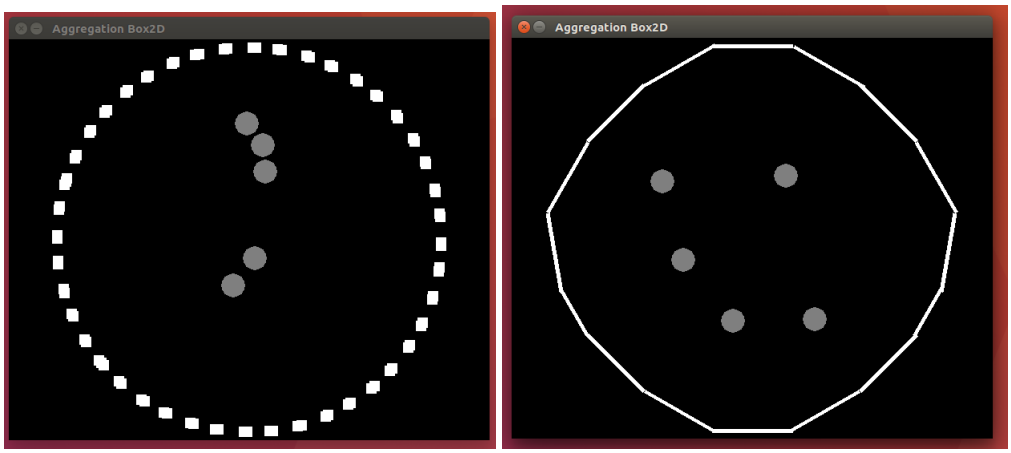


Figura 4.27: Escenario circular inicial para agregación en Box2D

El problema del escenario circular es que al tener 360 objetos estáticos, el renderizado se ralentiza considerablemente, por lo tanto, para evitar este problema se ha disminuido el número de paredes a 90 y se ha aumentado su tamaño a 0.3 x 0.3 unidades para evitar que los agentes puedan salir del círculo. Además, se ha aumentado la velocidad de los agentes con el fin de que la simulación se reproduzca al mismo ritmo que en el primer escenario. Sin embargo, el nuevo problema que encontramos es que puede que el ultrasonido no lea bien la distancia de los obstáculos porque puede detectar los huecos que hay entre las paredes.

El problema del escenario circular es que al tener 360 objetos estáticos, el renderizado se ralentiza considerablemente, por lo tanto, para evitar este problema se ha disminuido el número de paredes a 90 y se ha aumentado su tamaño a 0.3 x 0.3 unidades para evitar que los agentes puedan salir del círculo. Además, se ha aumentado la velocidad de los agentes con el fin de que la simulación se reproduzca al mismo ritmo que en el primer escenario. Sin embargo, el nuevo problema que encontramos es que puede que el ultrasonido no lea bien la distancia de los obstáculos porque puede detectar los huecos que hay entre las paredes.

Como solución al problema del sensor ultrasónico se ha sustituido el anterior círculo por un polígono de 16 caras. De esta forma no se satura la simulación y el agente puede detectar las paredes sin problemas. Además, la velocidad a la que actúan es la misma que en la del primer escenario.



(a) Segunda prueba de escenario circular para agregación en Box2D

(b) Escenario circular definitivo para agregación en Box2D

Figura 4.28: Escenario circular en Box2D

Por último, se ha creado un escenario con forma de cruz. Este tiene 12 paredes, por lo tanto, no tendrá el problema del anterior escenario, ya que la simulación no se sobrecarga de objetos estáticos.

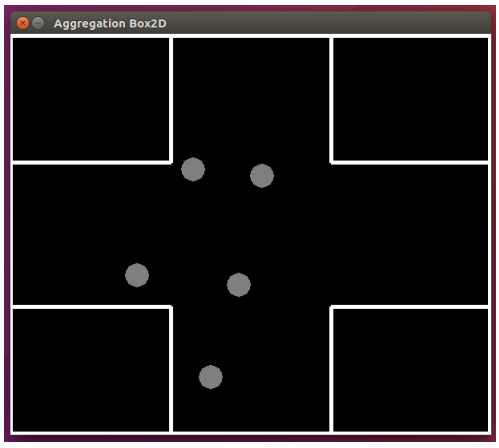


Figura 4.29: Escenario en forma de cruz para agregación en Box2D

4.5. Hardware utilizado

A continuación, se muestra un resumen del hardware utilizado para el desarrollo del proyecto.

CPU	Intel® Core™ i7-6500U 4 núcleos (8 hilos)
GPU	NVIDIA® GeForce® GTX 950M
RAM	32 GB DDR4

Tabla 4.2: Especificaciones hardware

5. Prueba de la plataforma robótica

El desarrollo de este capítulo es una toma de contacto con la plataforma robótica y tiene la finalidad de conocer su funcionamiento en una aplicación compleja como puede ser el equilibrio de un balancín. Otro de los objetivos de este capítulo es ver el comportamiento de varios robots en un entorno dinámico y como pueden desempeñar una tarea de forma conjunta. Para ello, primero se va a programar el equilibrio de un robot y después el de una pareja.

Para equilibrar un robot debemos tener en cuenta cuándo debe retroceder, avanzar o cambiar su velocidad, esto se realizará en función de su ángulo de inclinación en el eje horizontal. Además, para el desarrollo de este capítulo, la placa base tiene que estar colocada sobre el eje horizontal para poder leer correctamente los valores del giroscopio sobre el eje X.

Figura 5.1: Modelo del robot utilizado en la prueba de la plataforma robótica

5.1. Recogida y análisis de datos

Para saber si el robot tenía que avanzar o retroceder se leyó el giroscopio situando el robot en la parte inferior del balancín, donde tenía que avanzar, y después se colocó en la parte inversa, donde tenía que retroceder. Los datos que se obtuvieron fueron que debía avanzar si el grado leído era de signo negativo y se debía retroceder si era positivo. Para que el robot avance debe tener una velocidad positiva y para que retroceda debe ser negativa.

Un dato importante para tener en cuenta es que el robot puede leer del giroscopio 12 veces por segundo y puede cambiar la velocidad de los motores 3 veces por segundo, es decir, aproximadamente tarda en procesar la información del giroscopio 0.08 segundos y 0.33 segundos la de los motores. Por lo tanto, cada vez que se cambia la velocidad del motor perdemos 4 lecturas del giroscopio.

Otro dato para tener en cuenta es la velocidad mínima y máxima del motor, para que el motor se mueva debe tener una velocidad mínima de 22 RPM en una superficie plana y en

una superficie inclinada tiene que tener una velocidad mínima de 26 RPM. Por otra parte, la velocidad máxima que soporta el motor es de 255 RPM.

5.2. Equilibrio individual

Hemos implementado varios algoritmos para conseguir que el robot se equilibre en el balancín. Todos los algoritmos tienen el mismo esquema iterativo:

Código 5.1: Autoequilibrio en un balancín

```
1  while True:
2      x = ap.get_gyro_sensor_onboard("x")
3      v = get_speed(x)
4      set_speed(v)
```

Donde ***ap*** es el robot, ***x*** es el dato x del giroscopio, ***get_speed(x)*** es la función que calcula la velocidad a la que deben ir los motores según el valor de ***x*** y ***set_speed(v)*** es la función que asigna la velocidad ***v*** a los motores. Este algoritmo es reactivo, es decir, cada movimiento genera un estado en el balancín.

La tarea más complicada era saber la velocidad a la que debían ir los motores, ya que no podía ser muy alta porque el robot se salía del balancín y no podía ser menor que la velocidad mínima en una superficie inclinada porque el robot no se movería. Además, la velocidad debía variar en función del ángulo leído del giroscopio para poder encontrar un punto de equilibrio.

En el primer algoritmo se tenía en cuenta un historial de dos lecturas del giroscopio y la velocidad variaba en función de la diferencia entre el primer y último elemento de la lista. Además, la velocidad aumentaba al aumentar dicha diferencia. Los inconvenientes que tenía este algoritmo eran: que no tenía en cuenta una velocidad máxima del robot para que el robot no se saliera del balancín, al no disminuir la velocidad a tiempo el robot no encontraba un punto de equilibrio porque pasaba de un lado al otro de la tabla sin lograr estabilizarse en un punto y que al tener un historial de dos lecturas podía darse el caso en el que la primera lectura fuera en un lado del balancín y que la segunda y última lectura fuera del otro lado, por lo que la velocidad no era la adecuada en ese momento porque al ir muy rápido se pasaba ese punto de equilibrio y volvía a empezar.

El resultado se puede ver en el siguiente vídeo:
`https://drive.google.com/open?id=1kQob6ZXsZ3vHEaN4gxnfxm383diLLwjd`

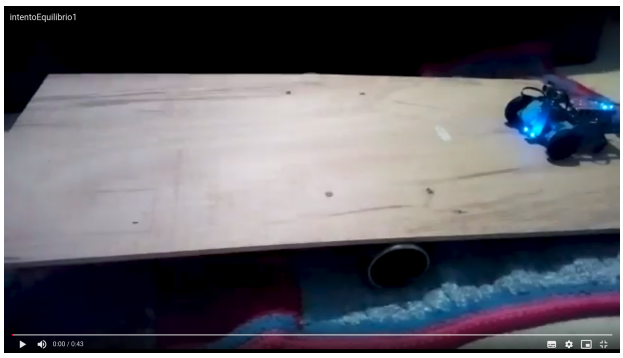
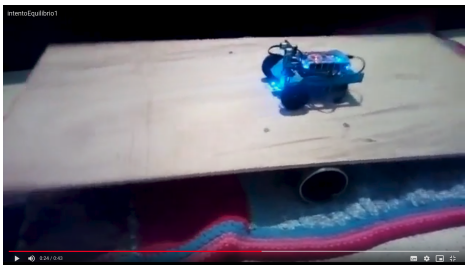


Figura 5.2: Instante inicial del primer algoritmo de equilibrio individual



(a) Instante en el que el robot encuentra el punto de equilibrio

(b) Siguiente movimiento del robot

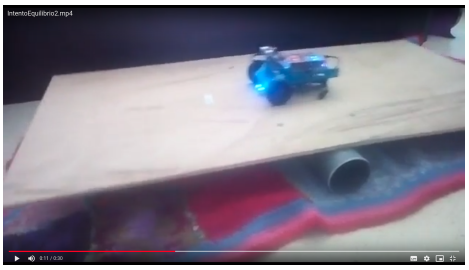
Figura 5.3: Resultados del primer algoritmo de equilibrio individual

Con este algoritmo el robot encuentra el punto de equilibrio, pero, debido al retraso de la lectura del giroscopio no detecta que lo ha encontrado a tiempo.

En el segundo algoritmo se aumentó el número de veces que se leía del giroscopio, es decir, se añadió un contador y la velocidad cambiaba cuando ese contador se agotaba o cuando hubiese un cambio de signo en el historial de lecturas del giroscopio. Además, había un máximo de velocidad para evitar los problemas del algoritmo anterior. Este algoritmo también presentaba los mismos inconvenientes que el anterior, aunque el robot en este caso iba más lento y se acercaba al punto de equilibrio, pero, no lo encontraba.

El resultado de este algoritmo se puede ver en el siguiente vídeo:
`https://drive.google.com/open?id=1a9jFHi1nv4f1esTDrbOHTLz1kJX8sJm_`

Figura 5.4: Instante inicial del segundo algoritmo de equilibrio individual



(a) Instante en el que el robot encuentra el punto de equilibrio

(b) Siguiente movimiento del robot

Figura 5.5: Resultados del segundo algoritmo de equilibrio individual

En el tercer y último algoritmo, se eliminó el historial de lecturas del giroscopio y la velocidad variaba en función de la lectura actual. Además, la velocidad mínima y máxima eran muy bajas para que el robot no se pasara el punto de equilibrio y pudiera cambiar la velocidad según en el punto en el que estaba.

La fórmula de la velocidad es la siguiente:

$$v = min(abs(x) \cdot 15, 55) \cdot sign(x) \cdot (-1) \tag{5.1}$$

Donde $\boldsymbol{v}$ es la velocidad a la que deben ir los motores, $\boldsymbol{x}$ es el ángulo leído del giroscopio y $\boldsymbol{sign}$ es la función que indica el signo de $\boldsymbol{x}$.

Si el ángulo de inclinación es menor de 4 grados, el robot disminuirá su velocidad y al alcanzar un grado o menos de inclinación el robot dejará de moverse porque la velocidad será menor que la velocidad mínima en una superficie inclinada. Además, si se desestabiliza, una vez encontrado el punto de equilibrio, vuelve a intentar autoequilibrarse para mantenerse en todo momento en equilibrio.

El resultado de este algoritmo se puede ver en el siguiente vídeo:
`https://drive.google.com/open?id=1o132XfLzV8tmGgqSsdP_QclhqO1o5zHh`

(a) Instante inicial del tercer algoritmo de equilibrio individual

(b) Robot equilibrado con equilibrio individual

Figura 5.6: Resultados del tercer algoritmo de equilibrio individual

5.3. Equilibrio con más de un robot

En este apartado se pretende que dos robots se autoequilibren en el mismo balancín del apartado anterior. Para ello se utilizarán distintas estrategias que se explicarán más adelante. Ambos robots son del mismo modelo y tienen la misma arquitectura aunque tengan piezas diferentes.

5.3.1. Equilibrio de forma individual

En primer lugar, para que dos robots se equilibren utilizamos el mismo programa con el que se equilibró un robot y lo ejecutamos de manera independiente en cada uno. Como resultado obtenemos que tarda más tiempo en encontrar un punto de equilibrio, ya que, ambos se mueven a la misma vez teniendo en cuenta solo su estado y cuando uno alcanza el equilibrio el otro se mueve sin tener esto en cuenta.

El resultado de este algoritmo se puede ver en el siguiente vídeo:
`https://drive.google.com/open?id=1Eqth-9avcBi6obRoY4BOSH4bpcXAiA0d`

(a) Instante inicial del algoritmo de equilibrio individual con más de un robot

(b) Robots equilibrados mediante equilibrio individual

Figura 5.7: Equilibrio de una pareja de robots mediante equilibrio indvidual

5.3.2. Equilibrio con sistema centralizado

En este apartado se ha implementado un algoritmo de control centralizado de ambos robots. En este algoritmo se tiene en cuenta tanto la velocidad con el estado de equilibrio de cada uno. Además, para evitar el problema del equilibrio de forma individual, es decir, para evitar que el movimiento de un robot desestabilice el equilibrio que ha conseguido el otro, se moverán de forma secuencial.

El funcionamiento del algoritmo es el siguiente:

Código 5.2: Equilibrio con sistema centralizado

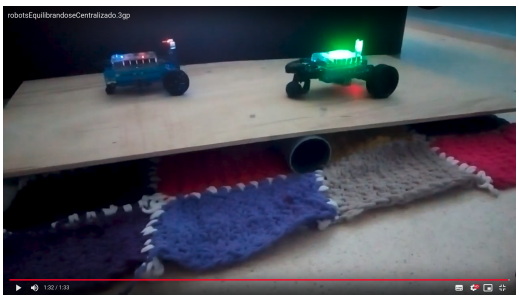
```
while True:
    x = ap.get_gyro_sensor_onboard("x")
    v = get_speed(x)
    ap.set_speed(v)
    sleep(0.5)   # esperamos a que se mueva una distancia mínima
    ap.set_speed(0)  # paramos el robot para ver el equilibrio que ha ↩
        ↪ generado

    sleep(1) # esperamos la reaccion del robot 1 para que actue el 2

    x = ap2.get_gyro_sensor_onboard("x")
    v = get_speed(x)
    ap2.set_speed(v)
    sleep(0.5)   # esperamos a que se mueva una distancia mínima
    ap2.set_speed(0)  # paramos el robot para ver el equilibrio que ha ↩
        ↪ generado

    sleep(1)
```

Donde ***ap1*** y ***ap2*** son el robot 1 y 2, respectivamente.

Como resultado obtenemos que ambos robots consiguen encontrar el punto de equilibrio de una manera más rápida que en el anterior apartado.

El resultado de este algoritmo se puede ver en el siguiente vídeo:
`https://drive.google.com/open?id=1GhclgZ8ASfBstz7oqx-g3Jtwhd2--bR_`

(a) Instante inicial del algoritmo de equilibrio con un sistema centralizado



(b) Robots equilibrados mediante sistema centralizado

Figura 5.8: Equilibrio de una pareja de robots mediante sistema centralizado

5.3.3. Equilibrio colaborativo

En este apartado se utiliza el estado de ambos robots para calcular un estado general y sobre este estado ambos robots ejecutarán la acción correspondiente.

El algoritmo se basa en el siguiente esquema:

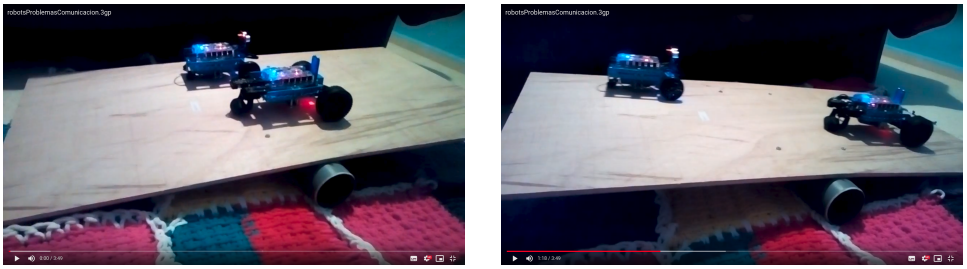
Código 5.3: Equilibrio colaborativo

```
while True:
    r1.read_state()
    r2.read_state()

    if r1.is_balanced() or r2.is_balanced():
        generalBalanced = True
    else:
        generalBalanced = False

    r1.action(generalBalanced)
    r2.action(generalBalanced)
```

Donde ***r1*** y ***r2*** son el robot 1 y 2, respectivamente y ***generalBalanced*** indica si hay un equilibrio general o no. El estado leído es el dato $\boldsymbol{x}$ del giroscopio y la acción que ejecutan es calcular la velocidad a la que deben ir los motores.

El inconveniente de este algoritmo es que al moverse ambos robots a la vez era más complicado encontrar un punto de equilibrio, ya que, al no estar sincronizados no se paraban a la vez y podrían desestabilizar el equilibrio que había generado el otro robot.

El resultado de este algoritmo se puede ver en el siguiente vídeo:
`https://drive.google.com/open?id=10oJ1l-zM1z1Z_HRUyF9sNlYq-OuycrDB`



(a) Instante inicial del algoritmo de equilibrio colaborativo

(b) Instante en el que se mueven ambos robots a la vez y no paran al encontrar el punto de equilibrio (minuto 1:18)

Figura 5.9: Equilibrio de una pareja de robots mediante equilibrio colaborativo

5.4. Conclusión

Finalmente, tras realizar este capítulo podemos concluir las ventajas y desventajas de los métodos utilizados para que una pareja de robots pueda equilibrar un balancín.

En primer lugar, el equilibrio individual presenta la ventaja de ser escalable, ya que, se pueden utilizar más o menos robots sin necesidad de cambiar el algoritmo inicial. Por otra parte, la desventaja de este método es que los robots solo tienen la información del entorno y no de las acciones ni del estado de los otros robots, por lo que en esta tarea era un inconveniente porque realizaban las acciones sin comprobar el estado que había generado el otro robot.

En segundo lugar, el equilibrio con un sistema centralizado presentaba la solución al problema de la estrategia anterior, ya que cada robot se movía después de comprobar el resultado de la acción del otro robot. En cambio, ofrecía otro inconveniente y es que al retirar el sistema central la tarea no se podía llevar a cabo. Por lo tanto, esta estrategia no era ni escalable ni robusta.

Por último, el equilibrio colaborativo es escalable y robusto, ya que la solución del problema no depende del número de robots utilizados. El problema que presentaba esta solución era que los robots tenían que sincronizarse para que un robot no desestabilizara el equilibrio generado por el otro.

6. Diseño de una conducta de agregación

En este capítulo se utilizará el método de diseño basado en el comportamiento para desempeñar una tarea de agregación. Esta tarea tiene como finalidad que un conjunto de robots se encuentren y se junten para en un futuro poder desempeñar otra tarea como enjambre. El motivo principal del desarrollo de este capítulo es poder realizar una tarea de enjambre sin aplicar inteligencia artificial con el fin de observar el comportamiento de los robots y las ventajas y desventajas de utilizar este método para el diseño de comportamientos de enjambre.

6.1. Sensores y actuadores utilizados

Para realizar esta tarea los datos se han discretizado para poder delimitar el espacio de búsqueda. A continuación se describen los datos utilizados:

Sensor de luminosidad: Este sensor únicamente indica si ha detectado luz o no, para ello devuelve un valor entero entre 0 y 1.

Sensor ultrasónico: La información de este sensor se ha dividido en 3 rangos de distancia: cerca, distancia media y lejos. Los valores que pertenecen a cada rango dependen del simulador que utilicemos, ya que, el tamaño de los escenarios en los simuladores no es la misma. El valor que devuelve varía de 0 a 2.

Sensor brújula: La información que nos interesa de este sensor es a qué punto cardinal está orientado el robot, es decir, si está apuntando hacia el Norte, Este, Sur u Oeste. El valor de este sensor varía de 0 a 3.

Velocidad: La velocidad se ha simplificado en 5 comandos simples como avanzar, girar hacia la izquierda, girar hacia la derecha, parar o retroceder, el valor abarca entre el 0 y el 4.

6.2. Algoritmo beeclust

El algoritmo de robótica de enjambre que se ha utilizado es una variante del algoritmo Beeclust (Schmickl y cols., 2016) y los datos de entrada son el sensor de luminosidad y el sensor ultrasónico, mientras que los datos de salida son las luces LED y el comando que indica la dirección que debe tomar. El algoritmo Beeclust está basado en el comportamiento de las abejas debido a que estas se reúnen en un lugar óptimo y mediante la detección de otras abejas. El funcionamiento del algoritmo es que el robot explora el entorno en busca de otros robots y para detectarlos utiliza el sensor de luminosidad, ya que la única diferencia entre un obstáculo y un robot es que el segundo tiene la posibilidad de emitir luz. Una vez encontrado un grupo de robots, este se une encendiendo sus luces LED y parando para no alejarse del grupo.

Todos los robots tienen el mismo comportamiento individual, pero, para poder indicar cuál es el primer robot que se une al grupo se encienden las luces de un robot al azar y el resto se

tienen que encargar de encontrarlo y unirse a él. Los movimientos para explorar son: avanzar, ir hacia la derecha o ir hacia la izquierda. El movimiento a ejecutar en la exploración se elige al azar, aunque, para facilitar la búsqueda del enjambre, si el robot detecta un nivel de luz más alto irá hacia el foco de luz. Esto quiere decir que si el sensor de luminosidad de la izquierda detecta más brillo que el de la derecha, el robot irá hacia la izquierda y viceversa.

Figura 6.1: Diagrama de transición de estados del algoritmo Beeclust

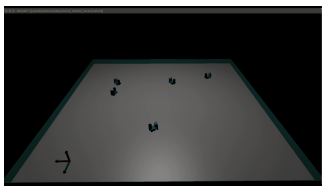
Para comprobar el funcionamiento de este algoritmo se han utilizado el simulador MORSE y los robots mBot Rangers reales. Además, se han utilizado los tres escenarios del simulador 3D relacionado con el enjambre de cinco robots.

6.3. Resultados obtenidos

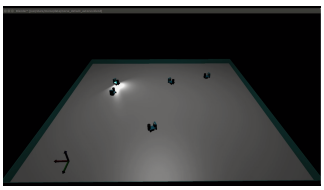
A continuación se comentan los resultados obtenidos en cada uno de los escenarios.

En primer lugar, se ejecutó en el escenario de la superficie cuadrada. Al principio, los robots se mueven para poder calibrar la brújula, esto se realiza siempre que vayamos a utilizar la brújula en el simulador MORSE. Después de calibrar la brújula, uno de los robots enciende sus luces LED y el resto explora la superficie para poder encontrar un grupo, conforme vayan acercándose al robot cuyas luces están encendidas, van agrupándose y encendiendo sus luces a su alrededor para formar un grupo. En algunas ocasiones los robots detectan con cierto retraso la luz de su entorno o la distancia de los obstáculos, esto se debe a que la ejecución del comportamiento de cada robot es secuencial, es decir, primero se lee el estado de los sensores de un robot y luego se ejecutan las acciones correspondientes de dicho robot. Por lo tanto, al iterar este proceso en cada robot, la información de los sensores no se interpreta en el momento adecuado. Este problema no puede solucionarse porque el simulador no permite la paralelización de la lectura de los sensores, por lo tanto, se debe ejecutar el algoritmo de manera secuencial.

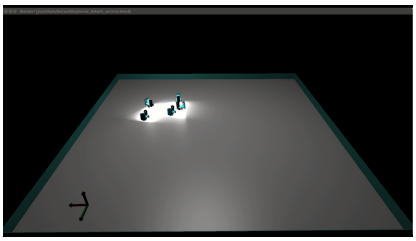
En el siguiente enlace está el video de la ejecución completa:
`https://drive.google.com/open?id=10Rs0_F3bKlZ0QldL5C2DZzZbC2tXwJXf`



(a) Inicio de la ejecución



(b) Primer robot que se une al grupo

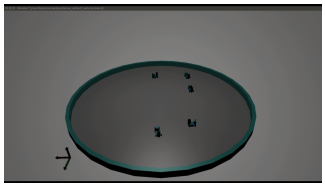


(c) Final de la ejecución del algoritmo y reunión de todos los robots

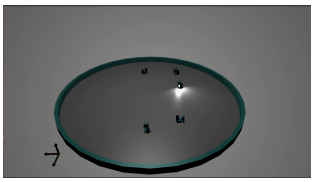
Figura 6.2: Ejecución del algoritmo beeclust en el escenario cuadrado de MORSE

En segundo lugar, se ejecutó el algoritmo en un recinto circular y el comportamiento de los robots es el mismo que en el del escenario cuadrado.

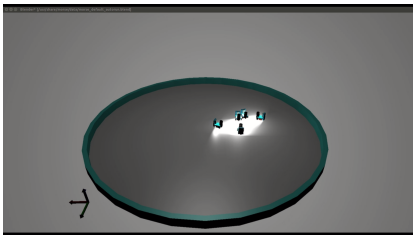
En el siguiente enlace está el video de la ejecución completa:
`https://drive.google.com/open?id=1G5pfdk0S-4vwPqio8KSIB7SEzFxu84vN`



(a) Inicio de la ejecución



(b) Primer robot que se une al grupo



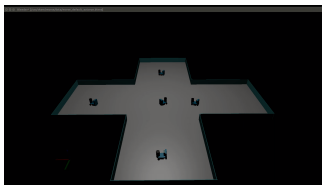
(c) Final de la ejecución del algoritmo y reunión de todos los robots

Figura 6.3: Ejecución del algoritmo beeclust en el escenario circular de MORSE

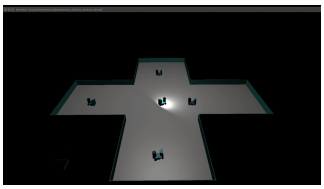
Por último, se ejecutó el algoritmo en un escenario con un recinto cerrado en forma de cruz. Este recinto tiene la dificultad de que cada robot se encuentra en una sección y todos deben explorar todo el escenario para encontrar un enjambre. Para comprobar el funcionamiento del algoritmo se han probado dos casos para la ubicación del robot que inicia el grupo: En el primer ejemplo, el primer robot se encuentra en el centro del escenario y en el segundo ejemplo, se encuentra en uno de los recintos.

En el primer ejemplo, los robots no tienen ninguna dificultad en encontrar el grupo, ya que, se encuentra en el centro del escenario y no necesitan explorar el resto de recintos.

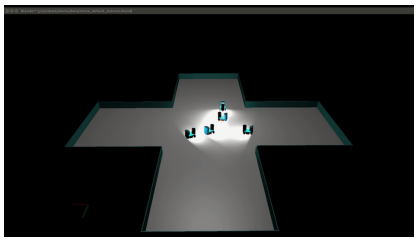
En el siguiente enlace está el video del comportamiento completo:
`https://drive.google.com/open?id=1ixFZPruWw_zzwza2Qq94JOP8ycLgJTE_`



(a) Inicio de la ejecución



(b) Primer robot que se une al grupo

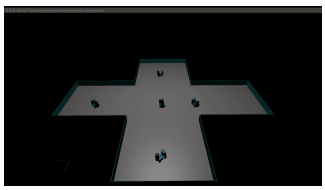


(c) Final de la ejecución del algoritmo y reunión de todos los robots

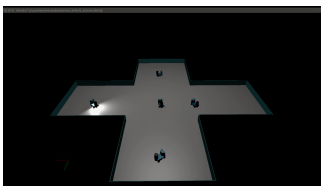
Figura 6.4: Primer ejemplo de ejecución del algoritmo beeclust en el escenario con forma de cruz de MORSE

Por otra parte, en el segundo ejemplo, podemos comprobar que los robots tardan más en encontrar al grupo porque tienen que explorar el resto de recintos. Por lo tanto, tardan más en encontrar el grupo en comparación con el ejemplo anterior, pero, finalmente consiguen agruparse.

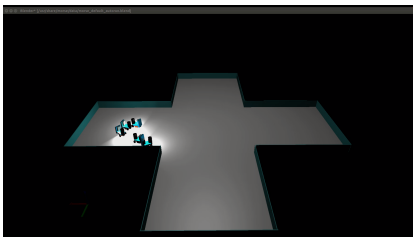
En el siguiente enlace está el video del comportamiento completo:
`https://drive.google.com/open?id=1uGuVmtikNMnWAGsls-ziIffUDaWzinBa`



(a) Inicio de la ejecución



(b) Primer robot que se une al grupo

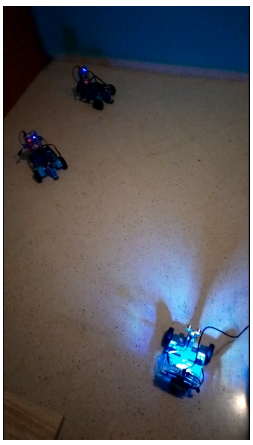


(c) Final de la ejecución del algoritmo y reunión de todos los robots

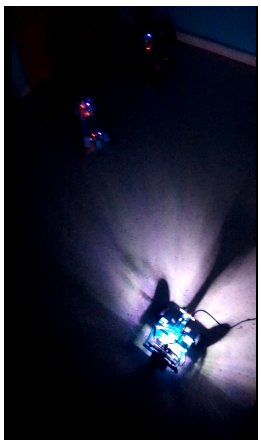
Figura 6.5: Segundo ejemplo de ejecución del algoritmo beeclust en el escenario con forma de cruz de MORSE

Finalmente, se han utilizado los robots mBot Ranger reales para comprobar el funcionamiento del algoritmo. Para que el sensor de luminosidad detectara solo la luz de los robots, se ha ejecutado con las luces apagadas. Se han utilizado 3 robots, aunque uno de ellos solo funcionaba con el cable USB, debido a que no se podía conectar mediante Bluetooth, por lo tanto, este robot se ha utilizado como inicio para que se agrupen el resto del enjambre. El funcionamiento del algoritmo no ha cambiado, pero, se ha tenido que modificar los datos del sensor ultrasónico y el sensor de luminosidad porque estos no tenían los mismos valores que en el simulador MORSE. El comportamiento de los robots ha sido similar al de los robots del simulador, ya que, evitaban obstáculos, seguían la luz y se unían al grupo una vez detectaban la luz suficiente.

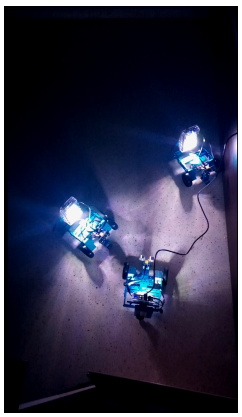
En el siguiente enlace está el vídeo del comportamiento completo:
`https://drive.google.com/open?id=1HSh560IQdAdcYfYcp6dEFp07ovyxbZ-B`



(a) Posición inicial de los robots



(b) Inicio de la ejecución sin luz ambiente



(c) Final de la ejecución del algoritmo y reunión de todos los robots

Figura 6.6: Ejecución del algoritmo beeclust en un entorno real.

6.4. Conclusión

Al finalizar la realización de este capítulo podemos concluir que el uso de este algoritmo no garantiza la agregación de todo el enjambre en todos los entornos, ya que, el robot deambula mientras busca al resto del enjambre y esto puede provocar que no llegue a encontrar el grupo. Además, otra desventaja de este algoritmo es que necesita que un robot encienda sus luces LED para que el resto se pueda unir a él, por lo que perdemos el uso de un robot y estamos limitando la zona de agregación a ese punto. Por lo tanto, hay que buscar otras políticas que permitan que un robot se una a un grupo sin que tenga que deambular por el escenario con movimientos aleatorios.

7. Aprendizaje de una conducta de agregación mediante Deep Reinforcement Learning

El motivo del desarrollo de este capítulo es encontrar una política óptima para el desarrollo de la tarea de agregación utilizando el método de diseño de comportamiento automático. Para ello, se van a comparar los resultados de diferentes estrategias evolutivas para el aprendizaje por refuerzo, distintas arquitecturas de la red neuronal o distinto número de robots que pertenecen al enjambre.

Los sensores y actuadores utilizados son los mismos que se han utilizado en el Capítulo 6, los únicos cambios que se han añadido en este capítulo son entorno a la velocidad. Una de ellas es la velocidad de giro, es decir, en este capítulo los giros hacia la derecha y hacia la izquierda serán de 90^{o} para evitar conflictos con las medidas de los ángulos entre el entorno simulado y el entorno real. La otra diferencia es que contamos con cuatro movimientos, ya que, no es necesario el comando para retroceder para la agregación de los robots.

La técnica de deep learning que se ha utilizado en este capítulo es la evolución de estrategias evolutivas con la librería Estool. Para obtener la función de recompensa nos hemos basado en el artículo SOYSAL y cols. (2007).

En este artículo utilizan la función de recompensa de la siguiente fórmula:

$$size(l)/n_{robots} \tag{7.1}$$

Donde ***size(l)*** es el tamaño del cluster con mayor número de robots y n_{robots} es el número total de robots.

Para obtener los clusters formados por los robots se obtiene la conexión considerando al grupo como un grafo dirigido y calculando la conectividad con el algoritmo de Floyd-Warshall, que se encarga de encontrar el camino mínimo en un grafo dirigido ponderado. Los nodos de este grafo son los robots y cada arista es la distancia que hay entre cada par. Sin embargo, no todos los robots están conectados, es decir, se considerarán las aristas de los nodos vecinos. Además, para calcular la distancia se utilizan las coordenadas (x, y) de la posición de cada robot.

Como esta función de recompensa es lineal, la hemos modificado para que premie las mejores soluciones y penalice las soluciones mediocres. Por lo tanto, la nueva función de recompensa es la siguiente:

$$f_{aggregation}(size(l)) = \begin{cases} (-10) & si \quad size(l) = 1/n_{robots} \\ size(l) \cdot 10 - 10 & si \quad size(l) < 0.5 \\ 10 \cdot size(l) & si \quad size(l) \geq 0.5 \end{cases} \tag{7.2}$$

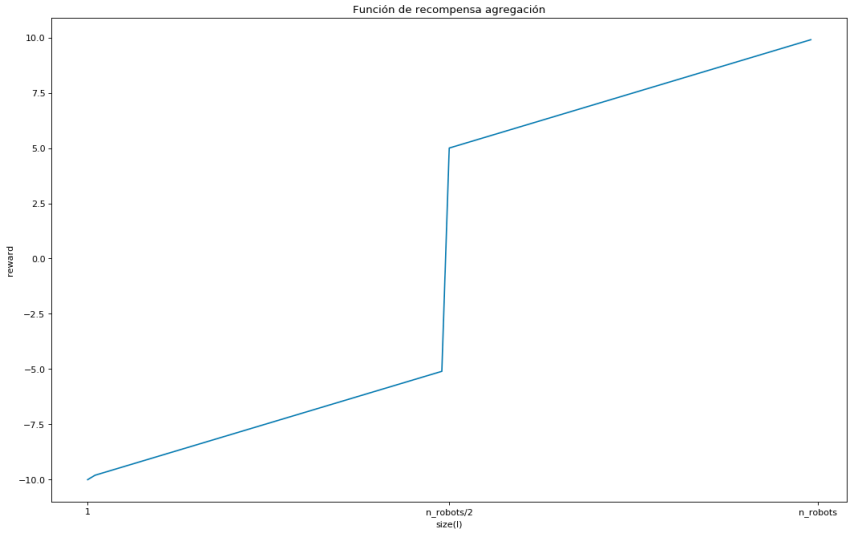


Figura 7.1: Función de recompensa de agregación

Las entradas de la red neuronal están discretizadas y transformadas en un array donde solo una de las opciones está activada. Por ejemplo, el sensor brújula tiene un array de 3 valores que pueden ser 0 o 1 y dependiendo del valor leído, es decir, si es 0, 1 o 2, el valor correspondiente a esa acción valdrá 1 y el resto 0. Además, todos los robots utilizan la misma red neuronal. Por lo tanto, esta servirá para cualquier número de robots. Además, para el entrenamiento de la tarea de agregación, los robots estarán posicionados lejos de los otros para evitar que la posición inicial influya negativamente en el proceso de aprendizaje.

En el proceso de aprendizaje se han utilizado diversas estrategias para poder mejorar el entrenamiento. Para determinar qué resultados son mejores se ha utilizado el test de Wilcoxon. Este test es una prueba no paramétrica para comparar el rango medio de dos muestras relacionadas y determinar si existen diferencias o no entre los resultados. Los valores de esta prueba se explican en el Anexo II y al valor ***pvalue/2*** lo denominaremos p-valor.

Por último, la capa de salida de la red neuronal está formada por 6 neuronas, donde 4 de ellas corresponden al comando de la velocidad y 2 a la acción de apagar o encender las luces LED. La función de activación de esta capa es softmax, por lo tanto, la salida de esta red neuronal es una clasificación de acciones que debe realizar el robot y la acción que se ejecuta es la que corresponde a la neurona activada.

7.1. Número de entradas de la red neuronal

Las salidas de esta red serán las mismas para todas las pruebas y son: el próximo comando de velocidad a ejecutar y el encendido o apagado de las luces LED. Los parámetros de entrenamiento en este apartado son: 32 redes neuronales de población por cada generación, 250 generaciones, CMA-ES como estrategia evolutiva, 5 robots/agentes para entrenar en un escenario, el escenario es una superficie cuadrada y la función de activación de las capas intermedias es tanh, tangente hiperbólica. Por útlimo, se ha evaluado cada política 3 veces y el resultado que obtenemos es la media del resultado de las 3 pruebas.

En primer lugar, se ha utilizado una red neuronal con cuatro entradas: Sensor ultrasónico, sensor de luminosidad, sensor de brújula y comando de velocidad ejecutado actualmente, aunque, como hemos comentado anteriormente, serían 4 vectores de entradas, por lo que tendríamos 13 entradas en total.

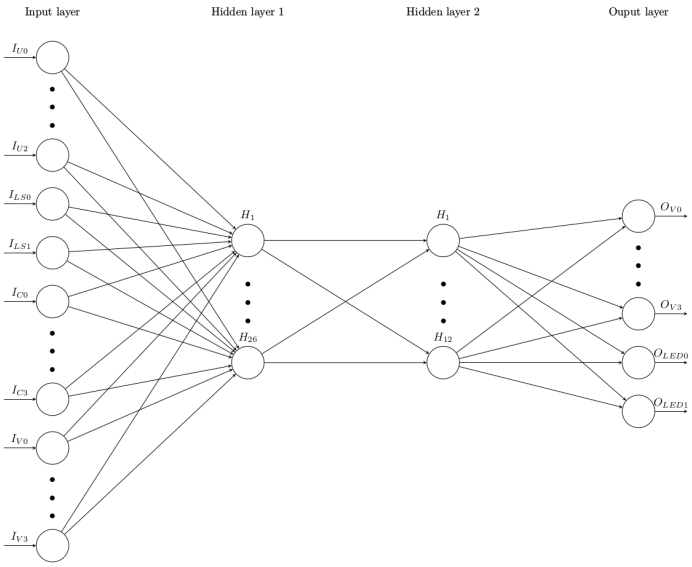


Figura 7.2: Red neuronal con cuatro entradas

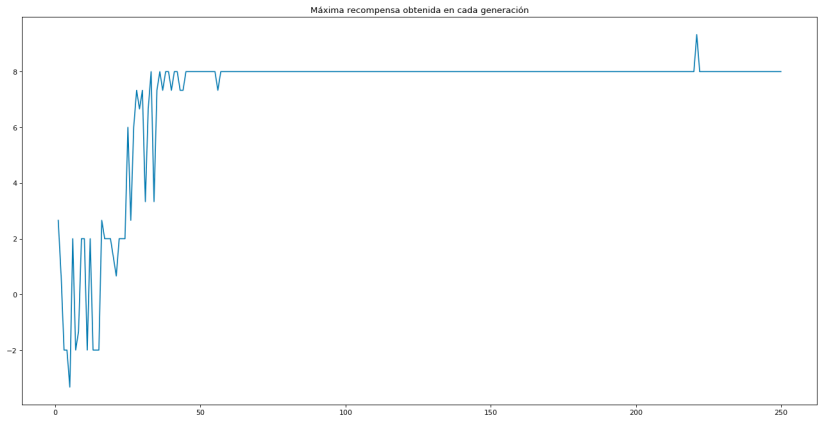


Figura 7.3: Resultados obtenidos por la red neuronal con 4 entradas

De la gráfica anterior podemos observar que la recompensa máxima alcanzada es 8, lo que equivale a un cluster de 4 robots. Por lo tanto, al final de la ejecución se ha agregado el 80% del grupo. Además, alcanza la mayor recompensa a partir de la generación 50 y, a partir de esta generación los resultados se estabilizan. El comportamiento resultante es que los robots se agrupan en la esquina superior de la derecha.

En segundo lugar, se ha probado a quitar de las entradas el comando de velocidad que ejecuta el robot en ese momento. Con esta modificación quedarían 3 entradas: sensor ultrasónico, sensor de luminosidad y brújula, y se modificaría el número de neuronas de la primera capa intermedia.

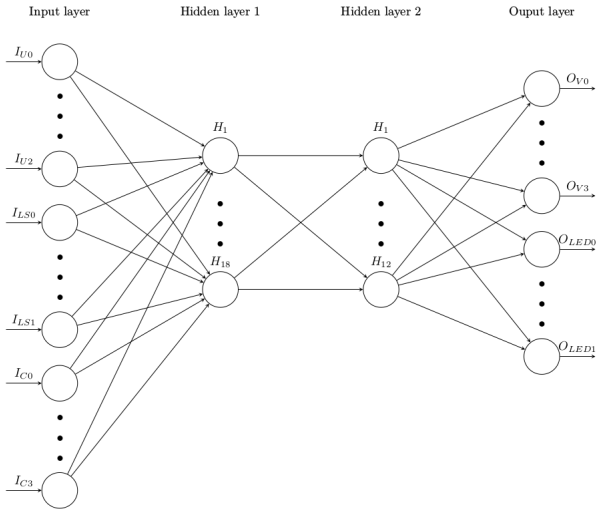


Figura 7.4: Red neuronal con tres entradas

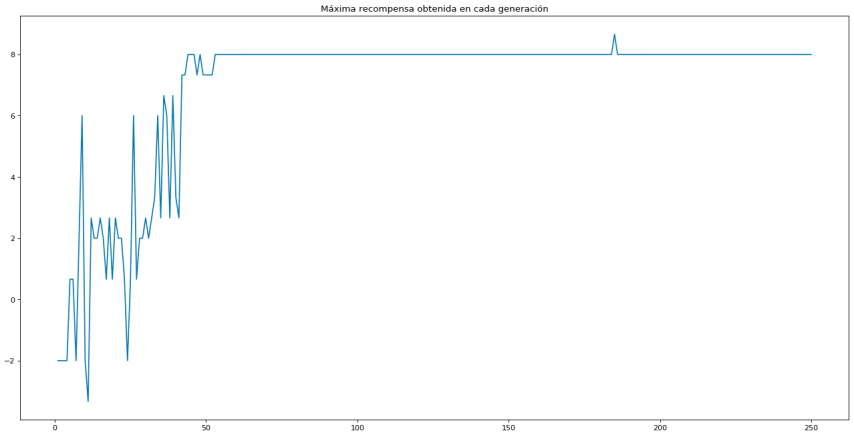


Figura 7.5: Resultados obtenidos por la red neuronal con 3 entradas

Los resultados obtenidos son muy parecidos a la red anterior, aunque en este caso, el entrenamiento oscila más antes de la generación 50, por lo que obtiene peores resultados que la primera red.

Por último, se eliminó la entrada correspondiente a los datos de la brújula con el fin de tener las mismas entradas del algoritmo Beeclust y poder comprobar si de esta forma se alcanzaba antes la solución óptima. Con esta nueva arquitectura las únicas entradas de la red neuronal eran el sensor ultrasónico y el sensor de luminosidad. Además, el comportamiento resultante es el mismo que el obtenido por la anterior red.

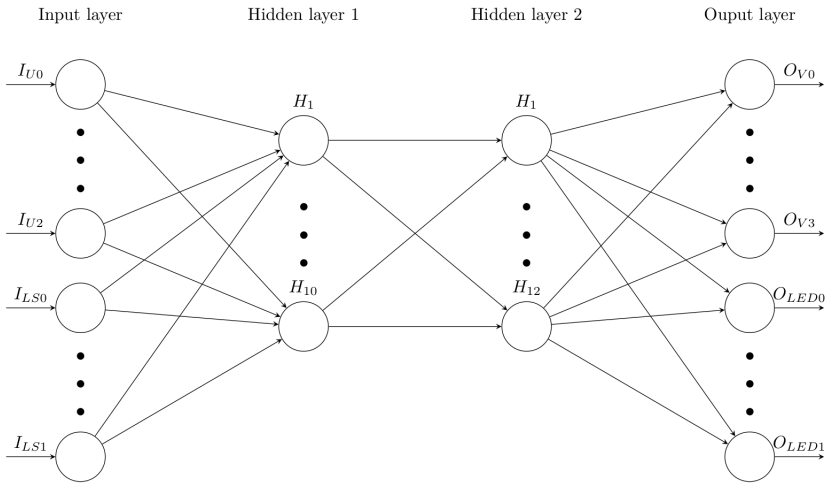


Figura 7.6: Red neuronal con dos entradas

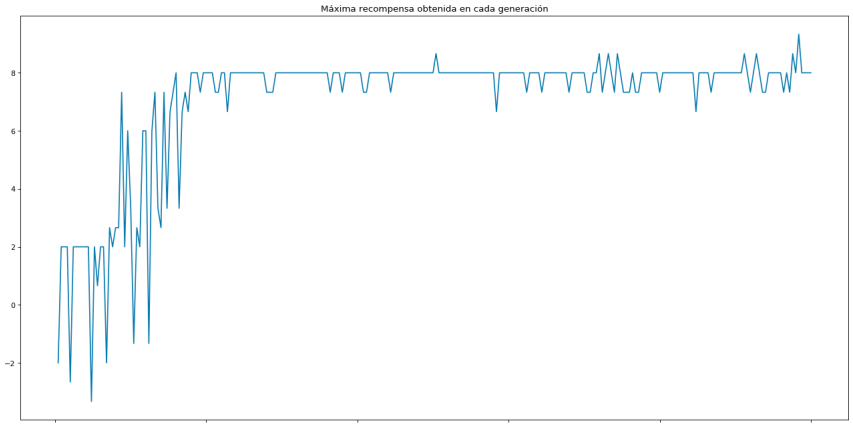


Figura 7.7: Resultados obtenidos por la red neuronal con 2 entradas

Los resultados obtenidos por esta red neuronal son menos estables que los obtenidos en las dos primeras redes. Esto es debido a que el comportamiento obtenido es que los robots se agrupan en distintas esquinas y no en la misma como pasaba con las otras dos redes neuronales, por lo tanto, no se llega a agrupar todo el enjambre.

Finalmente, al comparar los máximos resultados obtenidos en cada generación por las tres redes neuronales mediante un diagrama de cajas y bigotes obtenemos que la red que proporciona mejores resultados es la que esta formada por 4 entradas. Además, para confirmar que las recompensas obtenidas por la primera red son más altas que las obtenidas por las otras dos, se realizó el test de Wilcoxon comparando las tres arquitecturas. El p-valor obtenido comparando la primera y la segunda red es 0.0016 y el obtenido comparando la primera y tercera red es 0.0012, por lo tanto, al ser menores que 0.05 se rechaza la hipótesis nula y podemos afirmar que los datos obtenidos por la red neuronal de 4 entradas son más altos que los obtenidos por las otras redes.

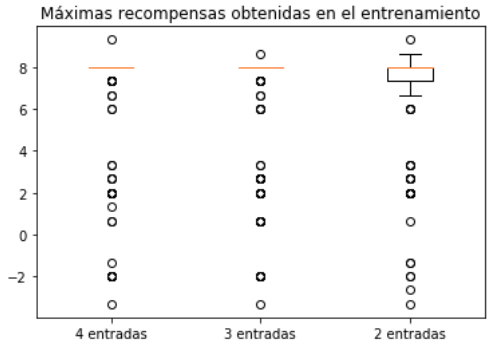


Figura 7.8: Comparación resultados con las diferentes entradas de la red neuronal

7.2. Estrategias evolutivas

En este capítulo se han estudiado todas las estrategias evolutivas que proporcionaba la librería Estool para poder comprobar cuál conseguía mejores resultados en 250 generaciones. Los parámetros de entrenamiento son los mismos que los del APARTADO 7.1 y la arquitectura de la red neuronal es la que contiene 4 entradas.

En primer lugar, se ha entrenado con la estrategia evolutiva **CMA-ES**, esta estrategia toma los resultados de cada generación y puede aumentar o disminuir el espacio de búsqueda para la siguiente generación con el fin de encontrar la política óptima. Los resultados y el comportamiento son los mismos que los obtenidos por la primera red neuronal del apartado anterior.

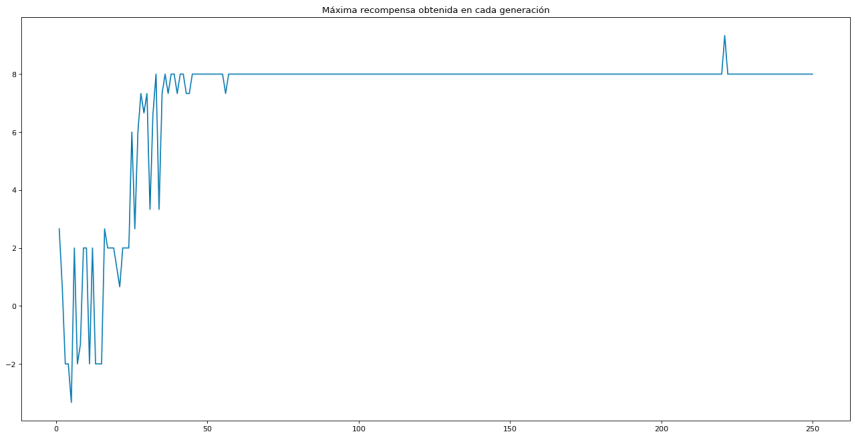


Figura 7.9: Resultados obtenidos por la estrategia evolutiva CMA

En segundo lugar, se ha entrenado con la estrategia evolutiva **PEPG**. En esta estrategia, la política está definida sobre una distribución sobre los parámetros de un controlador y estos se muestrean a partir de esta distribución al comienzo de cada ejecución. Los resultados de esta política no se estabilizan hasta la generación 150 y los mejores resultados de cada generación son menos estables.

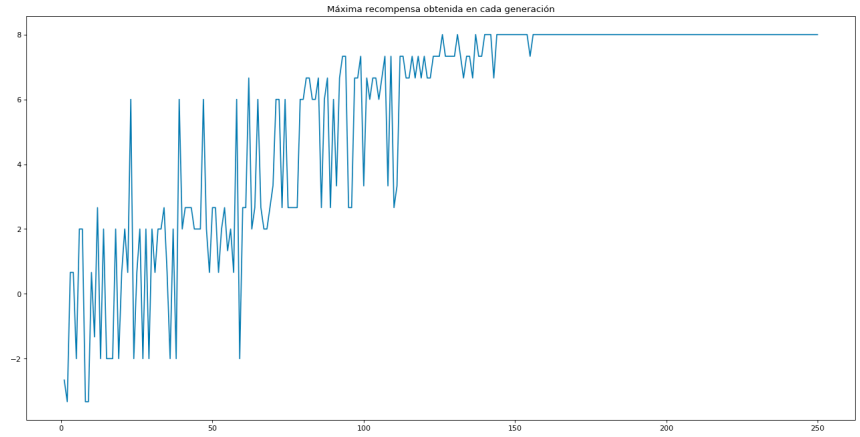


Figura 7.10: Resultados obtenidos por la estrategia evolutiva PEPG

En tercer lugar, se ha entrenado con **SES**, esta estrategia se basa en probar un conjunto de soluciones de una distribución normal con una media μ y una desviación estándar σ. Esta estrategia desecha todas las soluciones menos la mejor encontrada, lo que puede provocar que solo llegue a alcanzar un óptimo local, por lo que solo funciona en soluciones simples. Al utilizar esta estrategia se alcanza la recompensa máxima, es decir, se unen los 5 robots del enjambre, todos se unen en la esquina superior de la derecha.

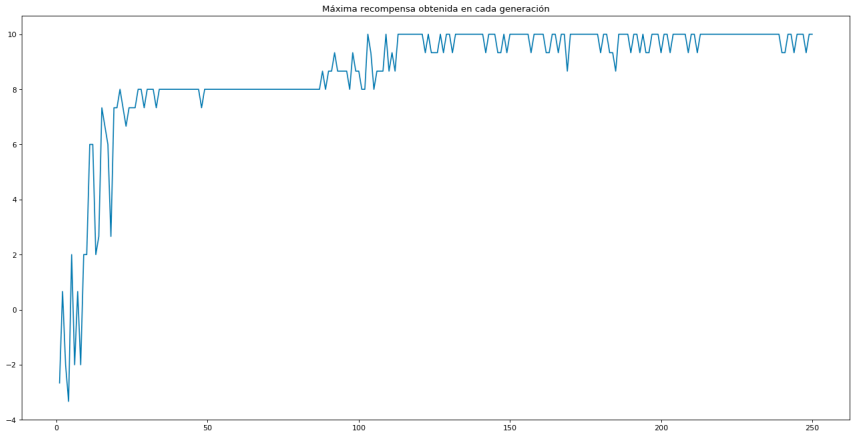


Figura 7.11: Resultados obtenidos por la estrategia evolutiva SES

En cuarto lugar, se ha entrenado con **GA**. Esta estrategia se basa en la teoría de las especies de Darwin, es decir, la primera generación crea una población de redes neuronales inicializadas al azar y para las siguientes generaciones, la población estará formada por las redes neuronales resultantes de del 10% de las mejores redes de la generación anterior, es decir, cada red neuronal de la población será un hijo de dos de las redes que han proporcionado mejores resultados. Con esta estrategia se llega a alcanzar la máxima recompensa, pero, los datos de entrenamiento oscilan más que los obtenidos por la estrategia SES.

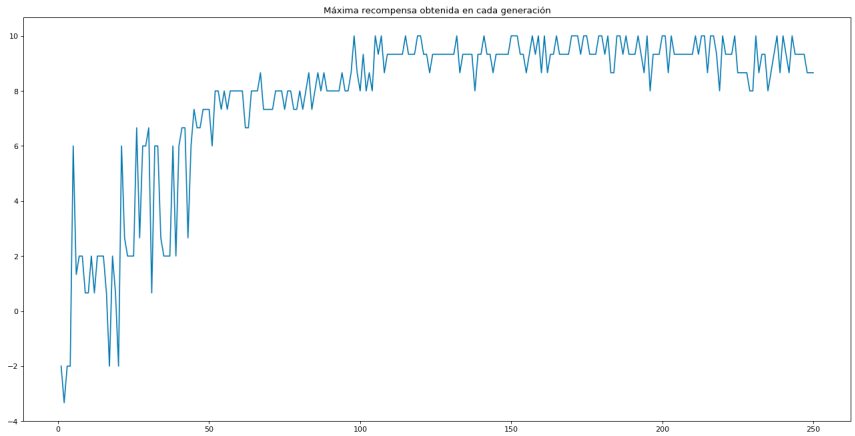


Figura 7.12: Resultados obtenidos por la estrategia evolutiva GA

Por último, se ha entrenado con la estrategia evolutiva **OpenES**. En esta estrategia se toma una distribución normal donde la desviación estándar σ es un número fijo y solo varía la media, μ, en cada generación. Como podemos observar en la siguiente gráfica la máxima recompensa en cada generación, a partir de la generación 150 es 8, por lo que solo se reúnen 4 de los 5 robots del enjambre. Además, el valor de las recompensas obtenidas por cada generación no se estabiliza hasta la generación 150, por lo que tarda más en alcanzar el mejor resultado que las estrategias CMA-ES o SES.

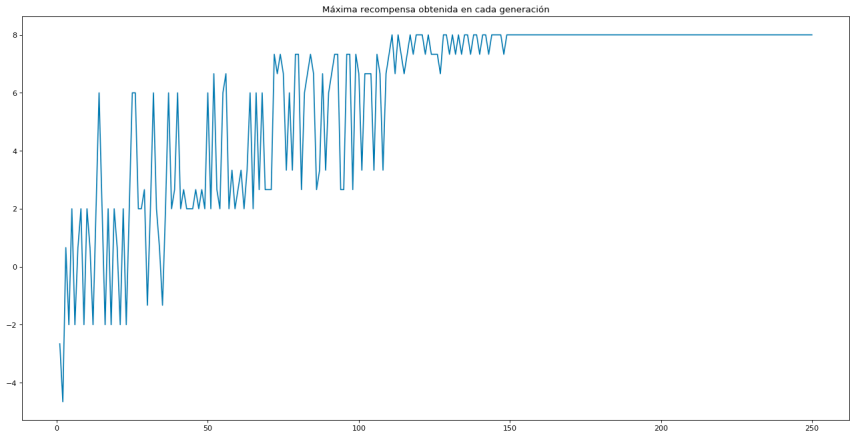


Figura 7.13: Resultados obtenidos por la estrategia evolutiva OpenEs

Para finalizar este apartado, se muestra una comparativa de los mejores resultados obtenidos en cada generación con las diferentes estrategias evolutivas. Al obtener este diagrama podemos observar que la estrategia
glsses obtiene mejores resultados que las otras cuatro estrategias. Para poder determinar si la estrategia
glsses obtiene recompensas más elevadas que el resto se ha utilizado el test de Wilcoxon y el p-valor obtenido en dicho test ha sido 0.0000 comparando todas las estrategias con SES, por lo que podemos afirmar que esta estrategia proporciona mejores resultados que las otras cuatro estrategias.

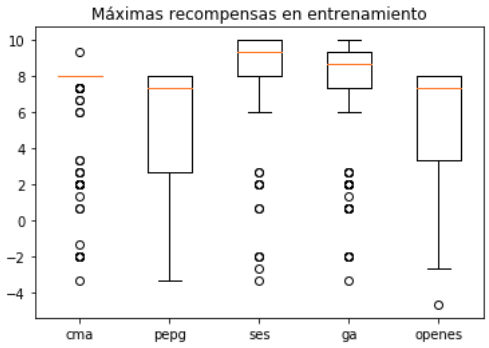


Figura 7.14: Comparación resultados con las diferentes estrategias evolutivas

7.3. Arquitectura de la red neuronal

En este apartado se ha modificado la arquitectura de la red neuronal con el fin de acelerar el proceso de aprendizaje y encontrar la política óptima. Para ello, se ha modificado el número de capas intermedias, el número de neuronas por capa y las funciones de activación de las capas intermedias. Todos los resultados se han obtenido en 200 generaciones y con los mismos

datos de entrenamiento del APARTADO 7.1.

7.3.1. Número de neuronas por capa intermedia

En este apartado probamos varias redes neuronales con distinto número de neuronas por capa intermedia, en las redes neuronales anteriores, el número de neuronas de la primera capa intermedia era el doble de las neuronas de la capa de entrada y el número de neuronas de la segunda capa intermedia era el doble de las neuronas de la capa de salida. La red neuronal original es la red cuyo número de neuronas de las capas intermedias es múltiplo de 2 y las redes modificadas son las que se explican a continuación.

La primera modificación de la red neuronal es aumentar el número de neuronas por capa de múltiplo de 2 a múltiplo de 5. Con esta modificación conseguimos alcanzar la máxima recompensa global a partir de la generación 75. El comportamiento resultante es que cada robot debía colisionar con una pared y girar hacia la derecha hasta alcanzar la esquina superior de la derecha, de esta forma se agrupaba todo el enjambre.

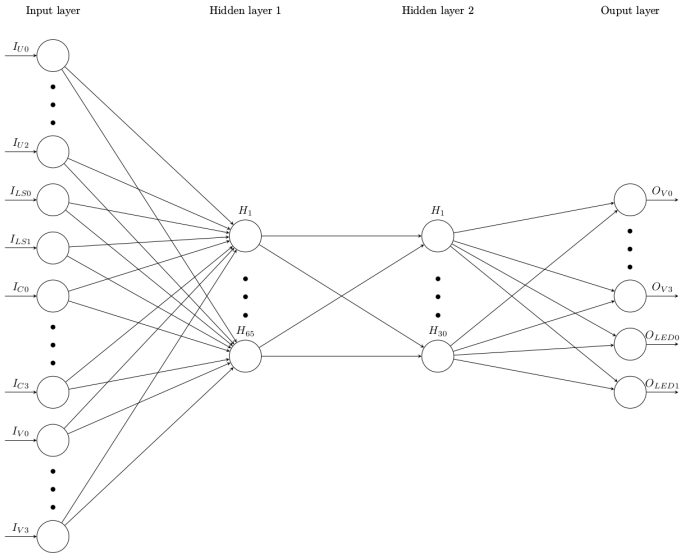


Figura 7.15: Red neuronal con número de neuronas de capas intermedias múltiplo de 5

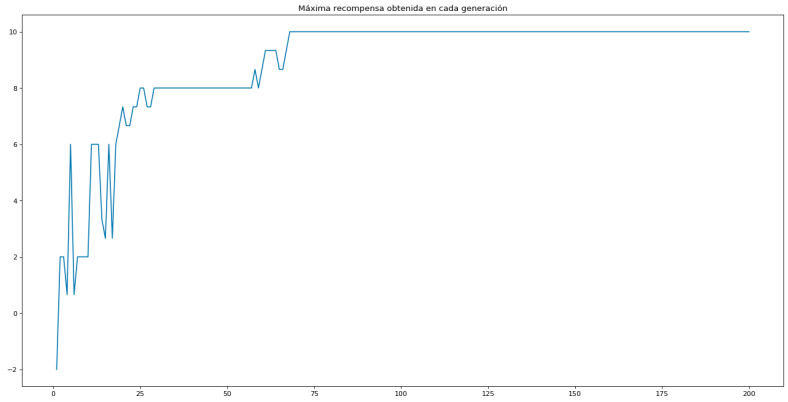


Figura 7.16: Resultados obtenidos por la red neuronal con número de neuronas de capas intermedias múltiplo de 5

La segunda modificación es disminuir el número de neuronas de las capas intermedias a múltiplo de 1. Los resultados del entrenamiento de esta red neuronal fueron más bajos que los obtenidos con la red neuronal anterior, ya que, el máximo alcanzado no era el máximo global.

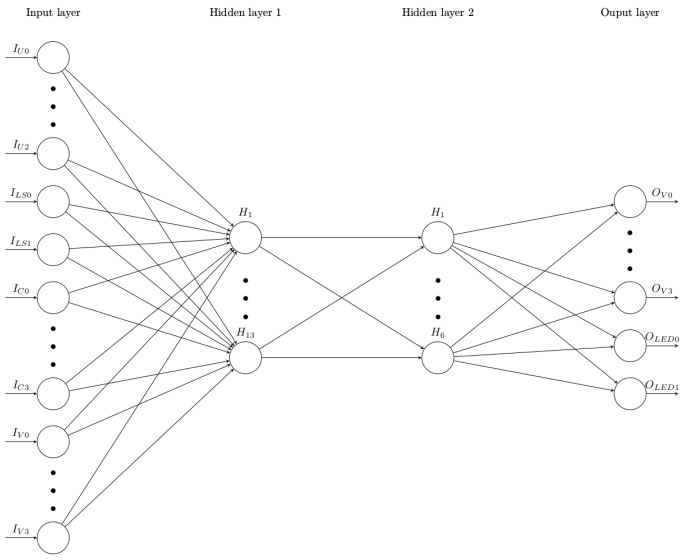


Figura 7.17: Red neuronal con número de neuronas de capas intermedias múltiplo de 1

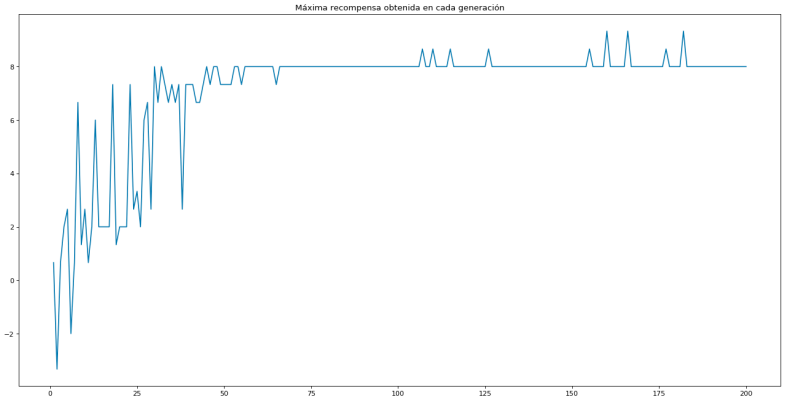


Figura 7.18: Resultados obtenidos por la red neuronal con número de neuronas de capas intermedias múltiplo de 1

Por último, se compararon los resultados mediante un diagrama de cajas y bigotes y mediante el test de Wilcoxon, el p-valor obtenido entre las muestras de la primera red modificada y con la red original es 0.0000 y el p-valor de las muestras de los resultados obtenidos por la segunda red modificada y los de la red original es 0.0740, por lo que podemos afirmar que la red neuronal que proporciona mejores resultados es la que contiene un número de neuronas de las capas intermedias múltiplo de 5.

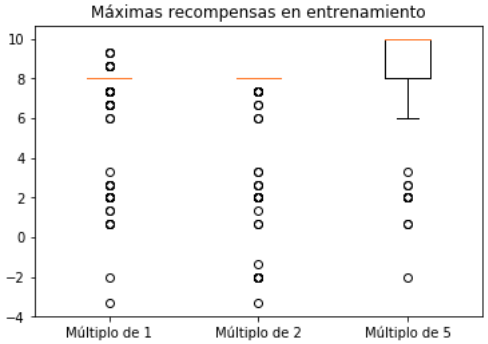


Figura 7.19: Comparación resultados con redes neuronales con diferente número de neuronas

7.3.2. Número de capas intermedias

Otra de las modificaciones de la red neuronal es quitar una de las capas intermedias para comprobar si al tener menos parámetros que explorar la red neuronal se aceleraba el proceso de entrenamiento y encontraba el máximo global. Para ello, quitamos la primera capa intermedia y no modificamos la segunda.

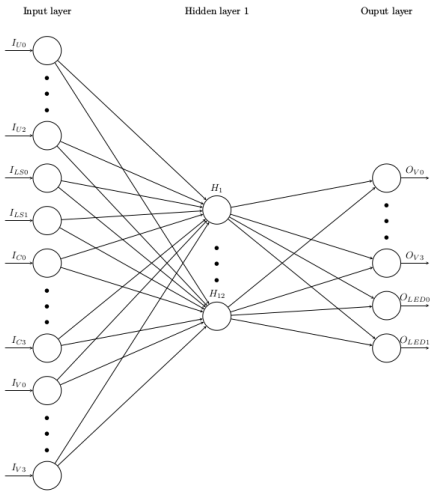


Figura 7.20: Red neuronal con 1 capa intermedia

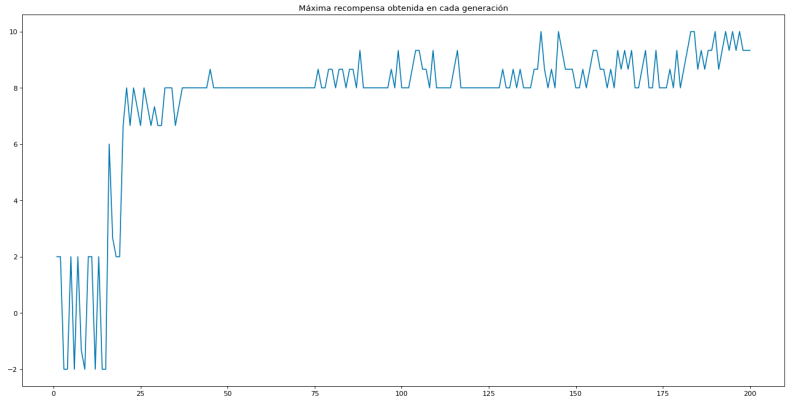


Figura 7.21: Resultados obtenidos por la red neuronal con 1 capa intermedia

Al comparar la red neuronal original con la red neuronal con una capa intermedia obtivimos que los resultados de esta eran más altos, ya que, el p-valor obtenido del test de Wilcoxon era 0.0000.

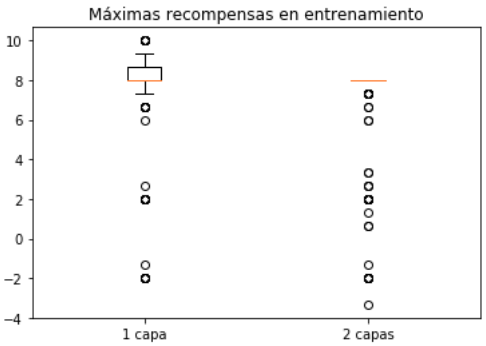


Figura 7.22: Comparación resultados con redes neuronales con diferente número de capas intermedias

7.3.3. Función de activación de las capas intermedias

Por último, se compararon varias funciones de activación para las capas intermedias: tanh, relu y sigmoid, para comprobar si alguna de ellas lograba acelerar el entrenamiento.

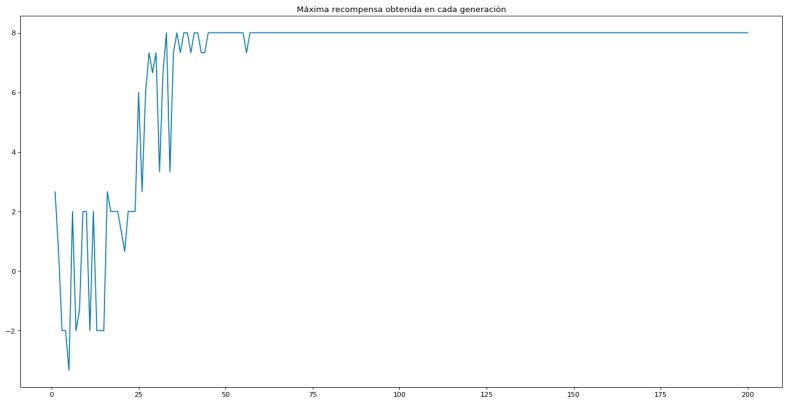


Figura 7.23: Resultados obtenidos por la red neuronal con función de activación tanh

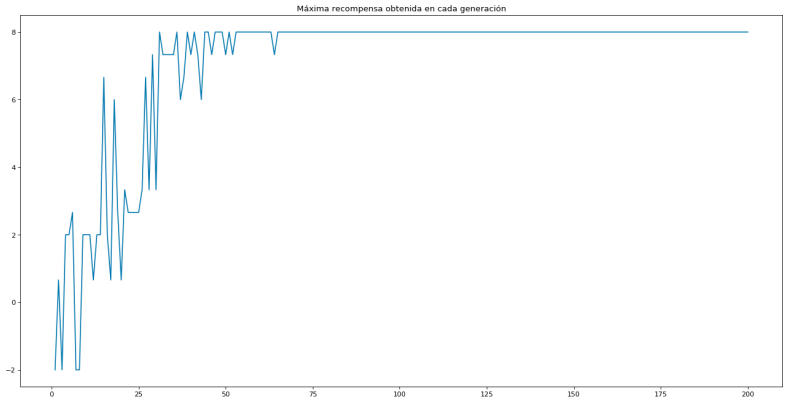


Figura 7.24: Resultados obtenidos por la red neuronal con función de activación relu

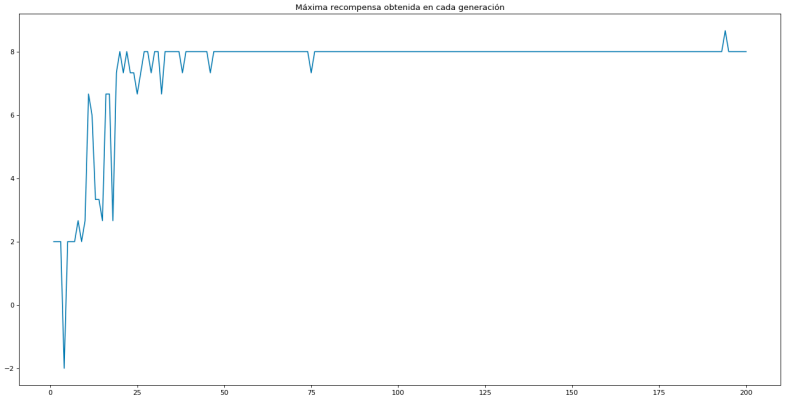


Figura 7.25: Resultados obtenidos por la red neuronal con función de activación sigmoid

Los resultados obtenidos son muy parecidos, con la diferencia de que los resultados de la red cuya función de activación era sigmoid proporcionaba datos ligeramente mejores. Al aplicar el test de Wilcoxon con los resultados de las redes con la primera y segunda función activación y con los resultados de las redes con la primera y tercera función de activación obtuvimos un p-valor de 0.1672 y 0.0000, respectivamente. Lo que, en un principio, podría indicar que la red neuronal con la función de activación sigmoid proporcionó mejores resultados que las redes con las otras dos funciones de activación, pero, esto se debe al valor que supera el 8 de la generación 194 y que en las siguientes generaciones no vuelve a aparecer. Por lo tanto, no podemos afirmar que ninguna funcion proporcione mejores resultados que las otras.

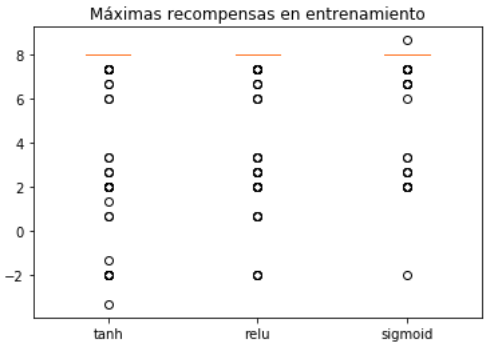


Figura 7.26: Comparación resultados con redes neuronales con diferentes funciones de activación de las capas intermedias

7.4. Escalabilidad

En este apartado se va a comprobar la escalabilidad de la red neuronal entrenada con 5 robots. Para ello, se va a utilizar la estrategia evolutiva SES, ya que, proporcionaba mejores recompensas en el entrenamiento. Además, la red neuronal constará de 4 entradas, dos capas intermedias de 65 y 30 neuronas en sus respectivas capas y la función de activación de estas capas será tanh, tangente hiperbólica. El escenario que se ha utilizado es el escenario con superficie cuadrada. El número de pasos total de ejecución son 4000 para que los robots puedan explorar el terreno.

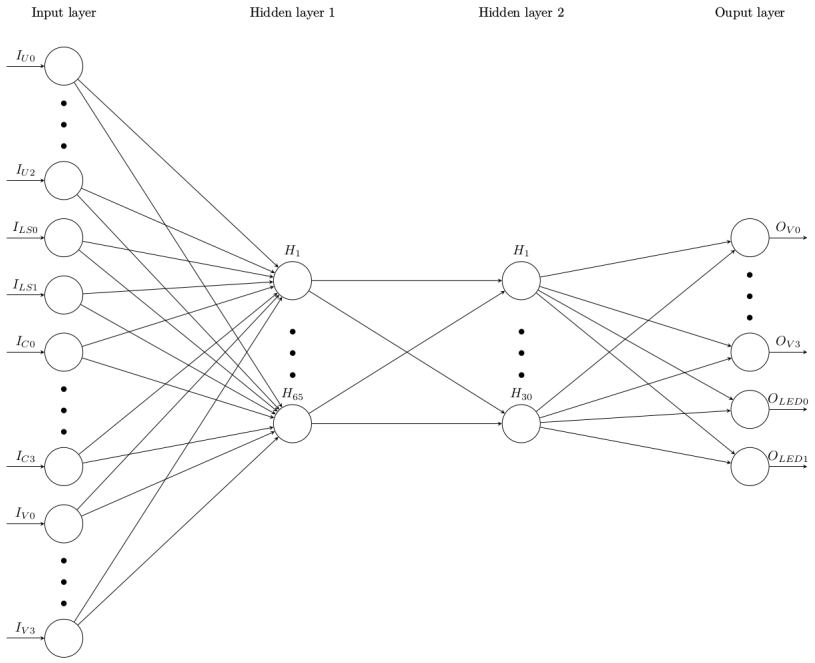


Figura 7.27: Red neuronal entrenada en el apartado de escalabilidad

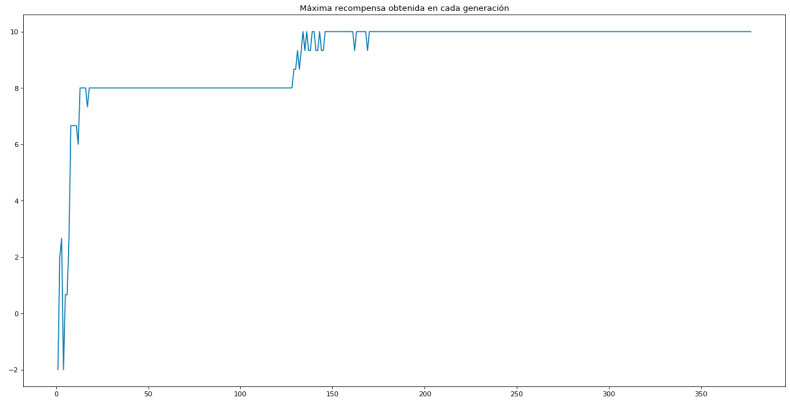
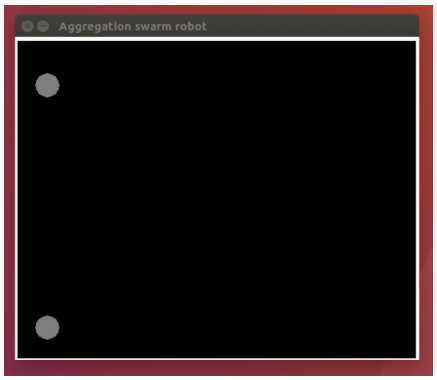


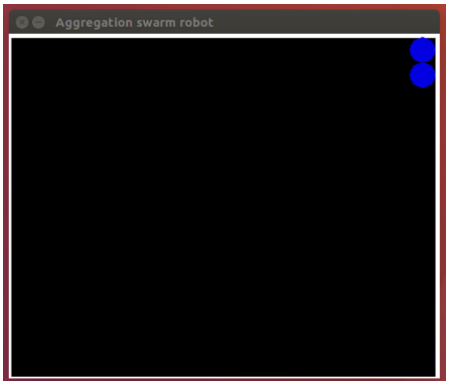
Figura 7.28: Resultados obtenidos por la red neuronal entrenada con 5 robots

En primer lugar, el comportamiento que obtenemos al utilizar esta red neuronal entrenada en un enjambre de 2 robots es que ambos se juntan en la esquina superior de la derecha, después se dirigen a la pared de la izquierda y no se separan de esa pared. Además, todos los robots encendían las luces LED la mayor parte del tiempo.

El vídeo de la ejecución completa se encuentra en el siguiente enlace: `https://drive.google.com/open?id=1pxf58Fr2cEl4RYYiuC4dOKWKXbX_puAC`

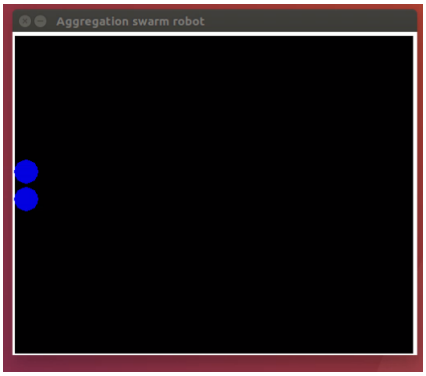


(a) Estado inicial del enjambre con 2 robots



(b) El enjambre se agrupa por primera vez

(c) Reorganización del enjambre

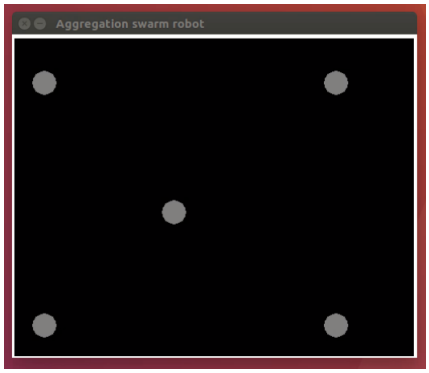


(d) Estado final del enjambre

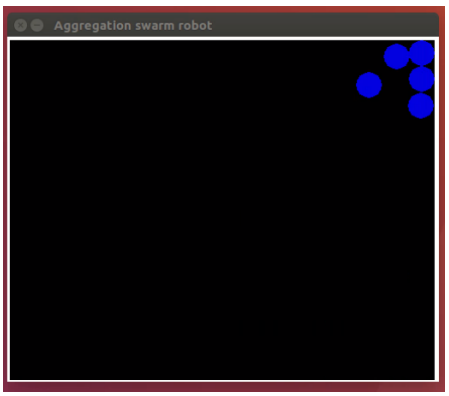
Figura 7.29: Agregación con 2 robots en 2D

En segundo lugar, ejecutamos la misma red neuronal en un enjambre de 5 robots y obtenemos el mismo comportamiento que en el anterior enjambre. El problema en esta ejecución es que no se juntan todos en la misma zona central de la pared y uno de ellos está separado, por lo tanto, se agrupa el 80% del enjambre.

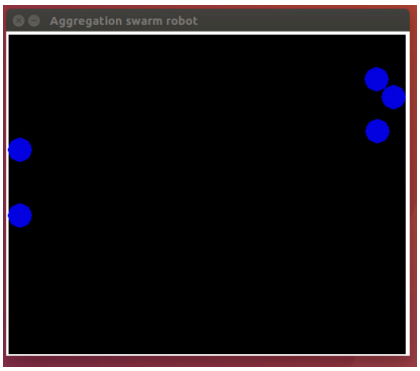
El vídeo de la ejecución completa se encuentra en el siguiente enlace: `https://drive.google.com/open?id=1aRvuROKWHnYn2M9cshfzQiTuZ3kx68ej`



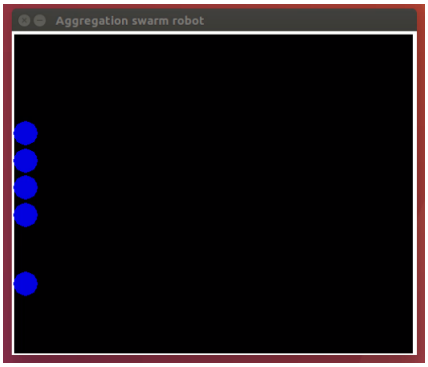
(a) Estado inicial del enjambre con 5 robots



(b) El enjambre se agrupa por primera vez



(c) Reorganización del enjambre

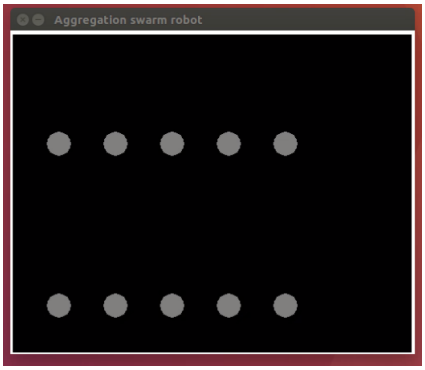


(d) Estado final del enjambre

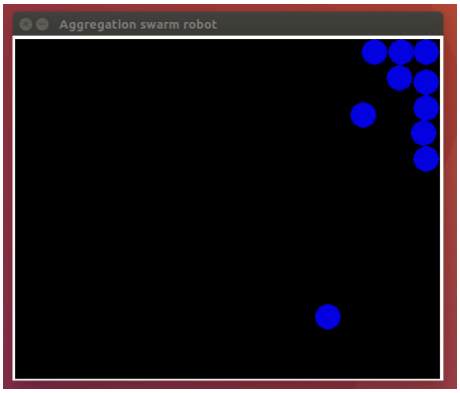
Figura 7.30: Agregación con 5 robots en 2D

En tercer lugar, ejecutamos la red neuronal en un enjambre de 10 robots y al final de la ejecución se agrupó todo el enjambre en la pared de la izquierda formando una fila.

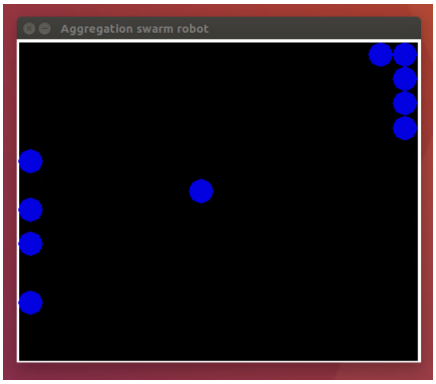
El vídeo de la ejecución completa se encuentra en el siguiente enlace: `https://drive.google.com/open?id=1YOgxlrqcKHDpJo8_RF-Q5JUDm6iqU4or`



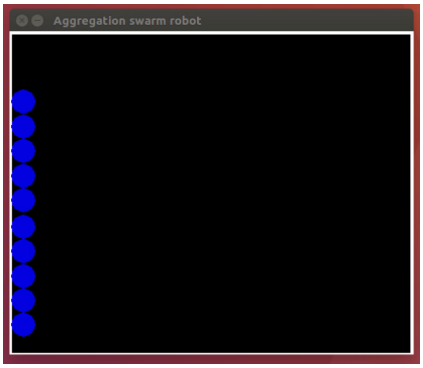
(a) Estado inicial del enjambre con 10 robots



(b) El enjambre se agrupa por primera vez



(c) Reorganización del enjambre

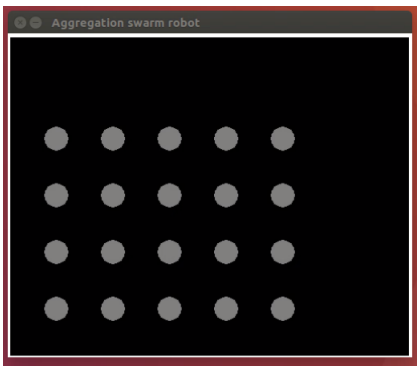


(d) Estado final del enjambre

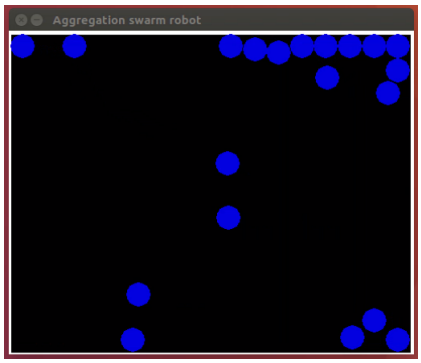
Figura 7.31: Agregación con 10 robots en 2D

En último lugar, ejecutamos la red neuronal en un enjambre de 20 robots y el enjambre se agrupó en la pared de la izquierda, como en las anteriores ejecuciones, con la diferencia de que en este caso todos los robots no se agruparon primero en la esquina superior de la derecha, como había pasado anteriormente.

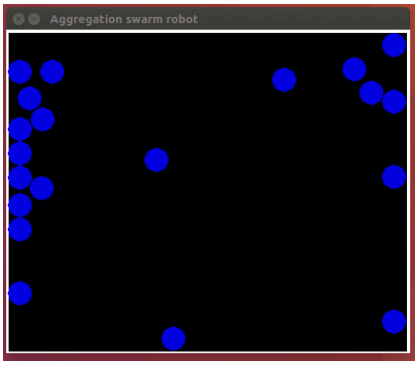
El vídeo de la ejecución completa se encuentra en el siguiente enlace: `https://drive.google.com/open?id=1RfEit6Aon3KdMxsaiPpVUkjFzcDeFIZR`



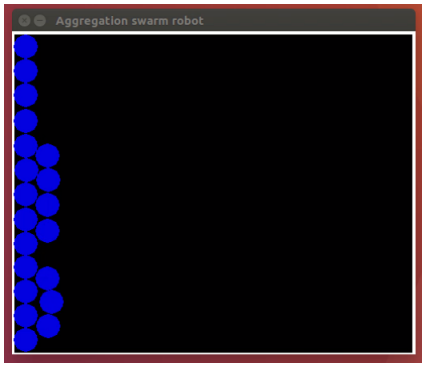
(a) Estado inicial del enjambre con 20 robots



(b) Parte del enjambre se agrupa por primera vez



(c) Reorganización del enjambre



(d) Estado final del enjambre

Figura 7.32: Agregación con 20 robots en 2D

Por otra parte, se ha realizado el entrenamiento con las mismas características que con la anterior red neuronal pero, con un enjambre de 20 robots para poder comprobar si funciona de la misma forma en el resto de enjambres.

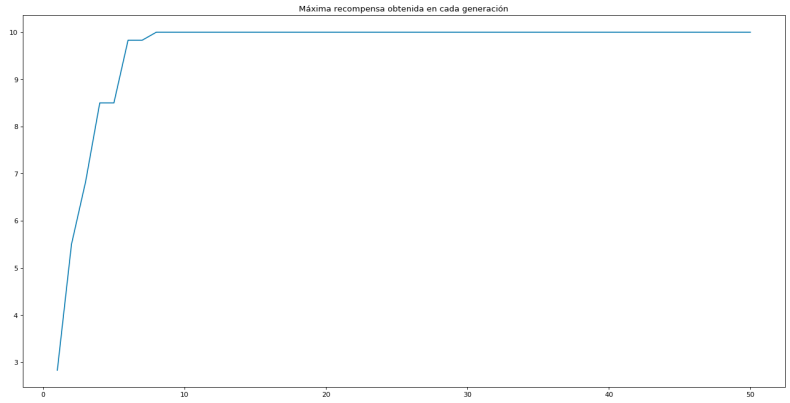
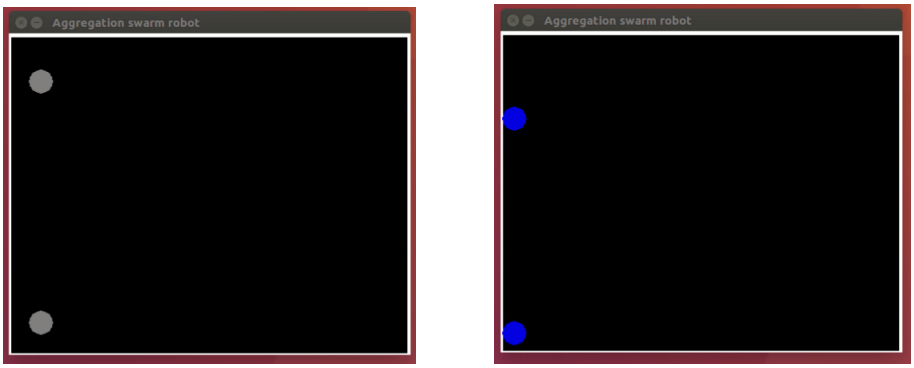


Figura 7.33: Resultados obtenidos por la red neuronal entrenada con 20 robots

En este entrenamiento se consiguió el máximo global a partir de la generación 10 y el comportamiento era que todos los robots giraban hasta encontrar el Oeste y después avanzaban hacia delante, de esta forma se agrupaban todos en la pared de la izquierda, también encendían las luces LED la mayor parte del tiempo. El problema estaba que al conectar esa misma red neuronal en otros enjambres con menor número de robots, el grupo no permanecía unido porque no se agrupaban en la zona central de la pared izquierda, sino que ocupaban toda la pared. Por tanto, este entrenamiento no es válido para el resto de enjambres.

A continuación, se muestran los resultados de las ejecuciones.

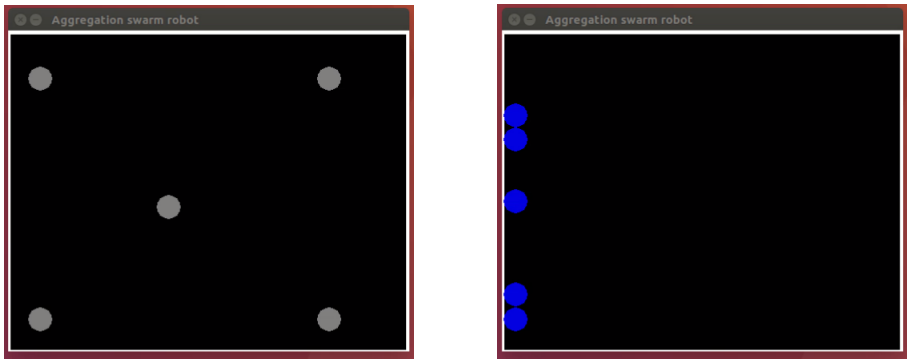
Comportamiento con 2 robots: `https://drive.google.com/open?id=148cSG2wr0ErpiHYyXDQd5xy_ajBzghtn`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.34: Agregación con 2 robots en 2D con una red entrenada mediante 20 robots

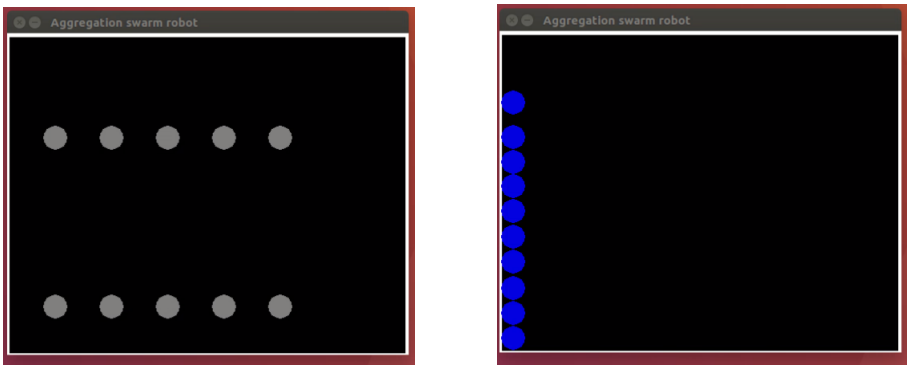
Comportamiento con 5 robots: `https://drive.google.com/open?id=1Kozrw-vGIjvTCKtsGDQt-vUHajs3Q7Sl`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.35: Agregación con 5 robots en 2D con una red entrenada mediante 20 robots

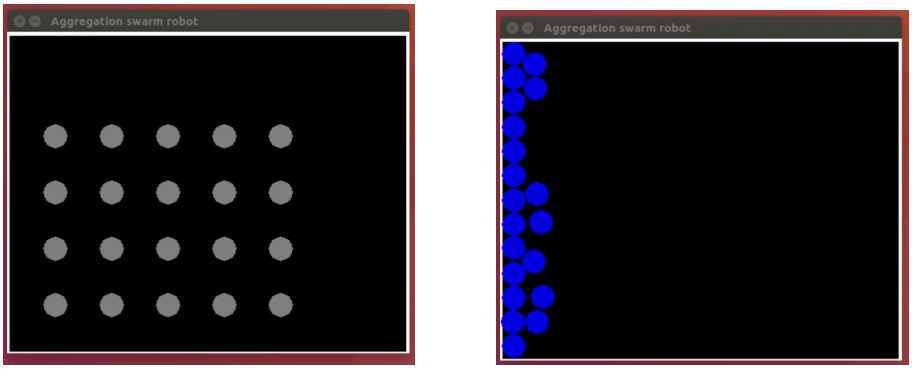
Comportamiento con 10 robots: `https://drive.google.com/open?id=12aj549xjCoMF-BD_BOkA3JuPhKZA1KyG`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.36: Agregación con 10 robots en 2D con una red entrenada mediante 20 robots

Comportamiento con 20 robots: `https://drive.google.com/open?id=1n2tRNj4Xs4Rh-aLeNbH5_m4aDYRGU6sz`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.37: Agregación con 20 robots en 2D con una red entrenada mediante 20 robots

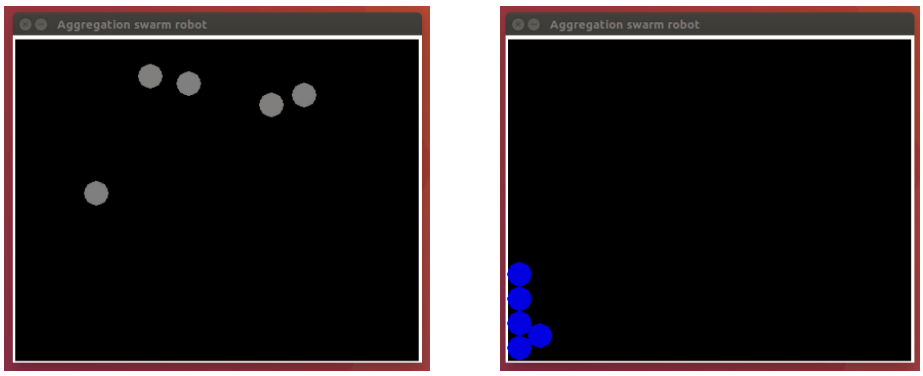
En conclusión, el entrenamiento de 5 robots es válido para enjambres de varios tamaños porque al aumentar el número de robots no hay problema para que se unan. En cambio, el comportamiento obtenido de entrenar 20 robots no es escalable porque conserva la política que mejor resultados proporcione con ese número de robots, aunque esta no sea la política óptima.

7.5. Resultados obtenidos en distintos escenarios

En este apartado se va a utilizar la red neuronal, entrenada para 5 robots, del apartado anterior con el fin de ver su funcionamiento en diferentes escenarios. Las posiciones de los robots se han generado al azar para generar un entorno distinto al del entrenamiento. Además, se han utilizado los simuladores 2D y 3D y los robots reales.

En primer lugar, se ha conectado al escenario de superficie cuadrada y el comportamiento era el mismo que en los apartados anteriores, es decir, primero giraban hacia el Sur, avanzaban hasta colisionar con una pared y después giraban hacia el Oeste y volvían a avanzar hasta colisionar con una pared. De esta forma, en el simulador 2D todos los robots se agrupaban en la esquina inferior de la izquierda.

Vídeo del comportamiento:
`https://drive.google.com/open?id=1yLXA4iJcJEsbGqztUYLyVFYBZMeuK1Zq`



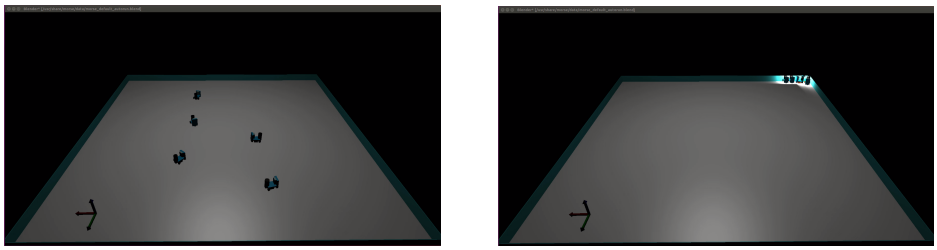
(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.38: Agregación en un escenario de superficie cuadrada en 2D

El comportamiento de los robots en el escenario de la superficie cuadrada en 3D ha sido muy parecido al obtenido en el entorno 2D, la diferencia ha sido que no han colisionado directamente con la pared situada hacia el Sur para colisionar con la esquina de la pared situada en el Oeste.

Vídeo del comportamiento:

`https://drive.google.com/open?id=1KrmJDkSesIv5px5sVo9gpbmcEcfwPB4A`



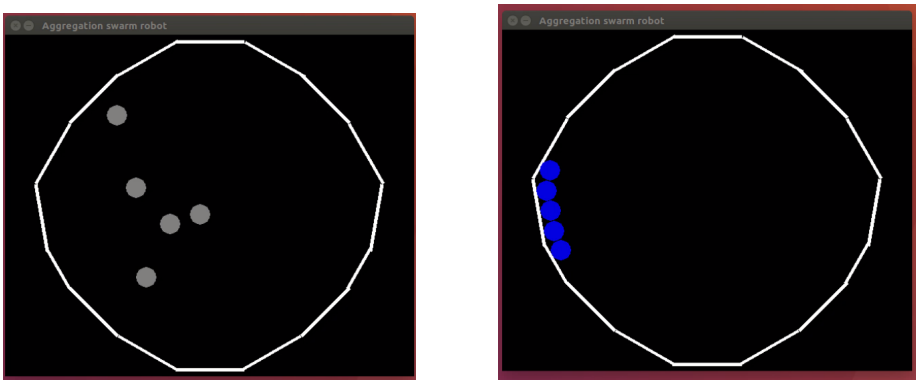
(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.39: Agregación en un escenario de superficie cuadrada en 3D

El comportamiento de los robots en la superficie circular del entorno 2D era el mismo que en la superficie cuadrada pero, al no haber esquinas los robots se han juntado en una de las paredes centrales situadas hacia el Oeste.

Vídeo del comportamiento:

`https://drive.google.com/open?id=1bd90B9Kg2KDgDKr5F009ENCzz8_A6zzA`

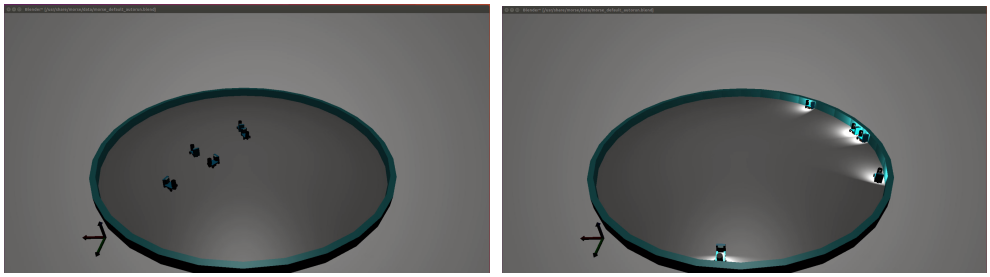


(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.40: Agregación en un escenario de superficie circular en 2D

Por otra parte, el comportamiento obtenido de los robots en la superficie circular del entorno 3D ha sido diferente porque al intentar colisionar con la pared del Sur, no han podido por lo que al final acababan navegando en círculos con la pared a su derecha. Por lo tanto, de esta forma no se ha podido agregar el enjambre.

Vídeo del comportamiento:
`https://drive.google.com/open?id=1Rp1XfaJ9G349w4PWYJ9ZGK8vGYGtj11J`

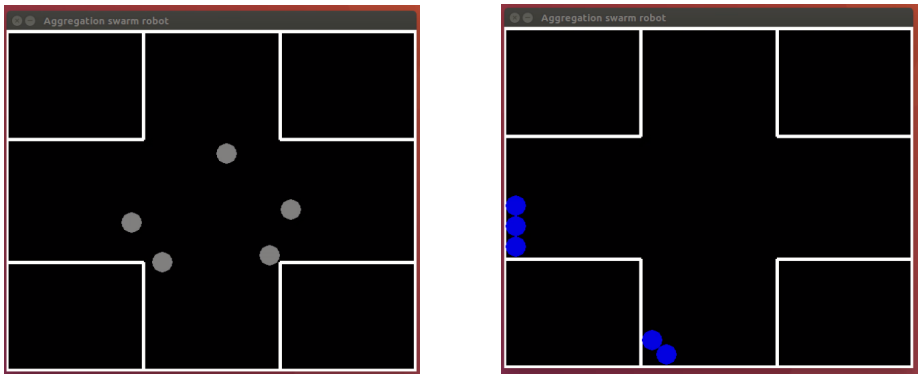


(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.41: Agregación en un escenario de superficie circular en 3D

En el último escenario, el escenario con superficie en forma de cruz, en el entorno 2D no se han podido agregar todos en la pared de la izquierda porque hay tres paredes en la zona de la izquierda, por lo tanto, se han dividido en 2 grupos, situados en dos de las paredes, y no se ha agrupado todo el enjambre.

Vídeo del comportamiento:
`https://drive.google.com/open?id=1s5RGf3_J1rr_xFiwpuqN5azLVhllzOOf`



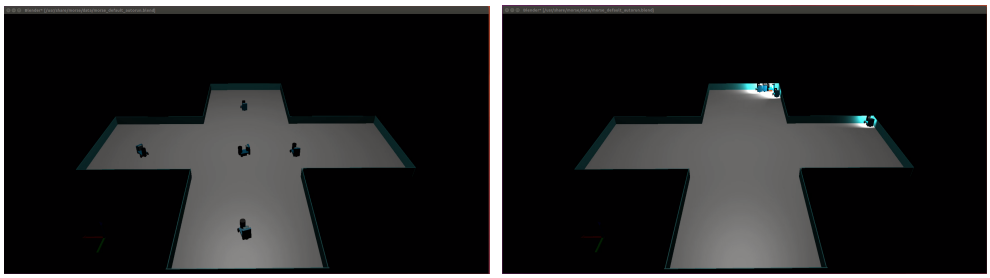
(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.42: Agregación en un escenario de superficie con forma de cruz en 2D

En el entorno 3D del escenario con forma de cruz se ha agrupado el 80% del enjambre, ya que, se han dividido en 2 grupos a la hora de recorrer el escenario.

Vídeo del comportamiento:

`https://drive.google.com/open?id=14hUmiblnaHNIexAdzD6WGueObAaYWkCV`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.43: Agregación en un escenario de superficie con forma de cruz en 3D

En el entorno real surgieron varios problemas, uno de ellos era al hacer un giro a la derecha o a la izquierda, ya que, los robots no leían el valor correcto de la brújula y acababan haciendo más giros para encontrar el grado exacto. Se intentó solucionar este problema disminuyendo la velocidad de giro lo máximo posible pero, en ocasiones, seguía habiendo problemas con la lectura de la brújula. Otro de los problemas es que al colisionar con la pared que estaba orientada hacia el Oeste acababan girando y leyendo que estaban situados al Sur, ya que, al chocar con esa pared acababan girando hacia la izquierda. Por lo tanto, no se podían quedar situados en esta pared porque no leían que estaban en el punto cardinal correcto.

Vídeo del comportamiento:

`https://drive.google.com/open?id=1WTMsq3vnfmtO1SHw2BZjTJt2sKQKMXe3`

(a) Estado inicial del enjambre

(b) Estado final del enjambre

Figura 7.44: Agregación en un entorno real

Por último, para evitar que se formaran dos grupos en la agregación en el escenario con forma de cruz se realizó el entrenamiento con este escenario y con 5 robots. En el entrenamiento se obtuvo como máxima recompensa 8, es decir, se agrupaba el 80% del enjambre y a partir de la generación 200 se obtuvo un mejor resultado. Sin embargo, en la evaluación del modelo las recompensas obtenidas eran valores muy cercanos a 8. Por lo tanto, se entreno el modelo en 230 generaciones porque no mostraba ninguna mejora conforme iba evolucionando.

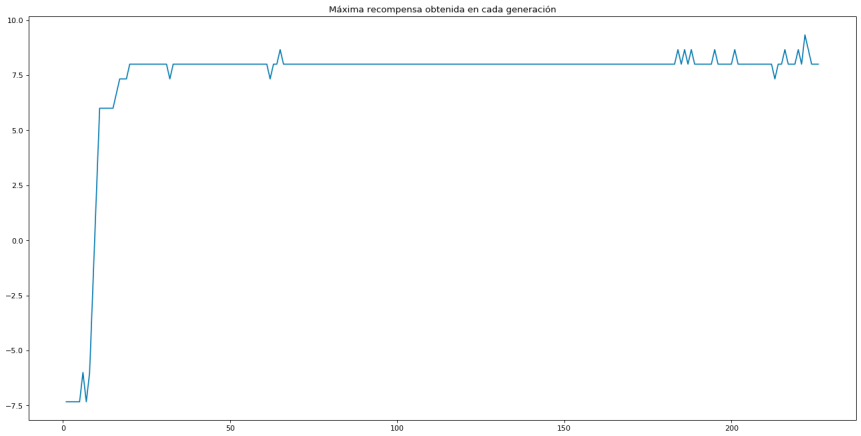


Figura 7.45: Resultados obtenidos en el entrenamiento en el escenario con forma de cruz

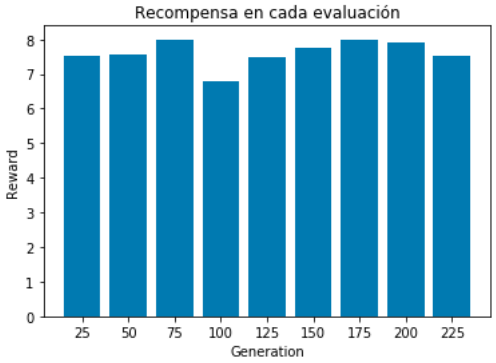
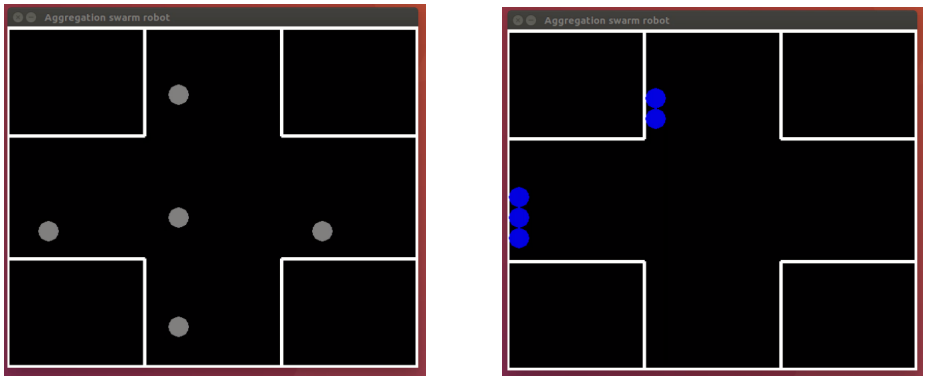


Figura 7.46: Resultados obtenidos en la evaluación durante el entrenamiento en el escenario con forma de cruz

El comportamiento obtenido en el entorno con forma de cruz, tanto en el entorno 2D como 3D, era distinto al obtenido anteriormente porque para dirigirse hacia la pared situada hacia el Oeste, giraban hacia la izquierda y avanzaban un paso, en vez de solamente avanzar cuando ya estuviesen orientados. En el entorno 2D se formaban dos grupos, por lo que no se agregaba el enjambre completo.

Vídeo del comportamiento:

`https://drive.google.com/open?id=1PlyyH3vNrzagiHVQDRfozzf9gLwfw0Z9`



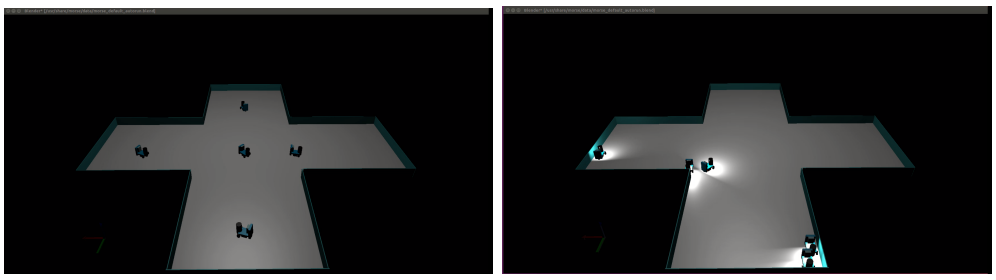
(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.47: Agregación en un escenario de superficie con forma de cruz en 2D con una red neuronal entrenada en un escenario con forma de cruz

En el entorno 3D tampoco se llegaba a reunir el enjambre en una misma localización. Una de las diferencias en el comportamiento de los robots que se ha obtenido en este entorno es que estos no se quedaban en una localización en concreto, como ocurría en el entorno 2D. Esto se puede deber a que las colisiones con otros robots perturbaran el comportamiento de estos o que en el comportamiento 2D no se pudieran haber movido por el contacto con los otros robots.

Vídeo del comportamiento:

`https://drive.google.com/open?id=1kyg7Twr28Z7JZwFnF7hESmnre9jtLxF8`

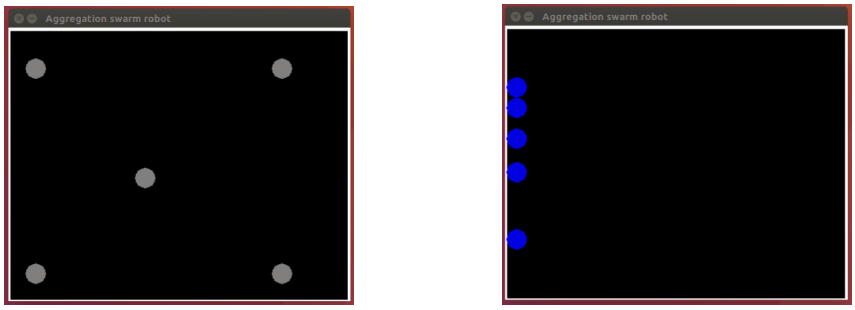


(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.48: Agregación en un escenario de superficie con forma de cruz en 3D con una red neuronal entrenada en un escenario con forma de cruz

Para comprobar si este comportamiento podía proporcionar mejores resultados en un escenario más simple se ejecutó en los escenario cuadrados de los entornos 2D y 3D. El comportamiento resultante no fue el esperado, ya que, tampoco se lograba reunir al enjambre completo y los robots que se unían en el entorno 3D lo hicieron por una colisión con otro robot que no les permitía realizar más movimientos.

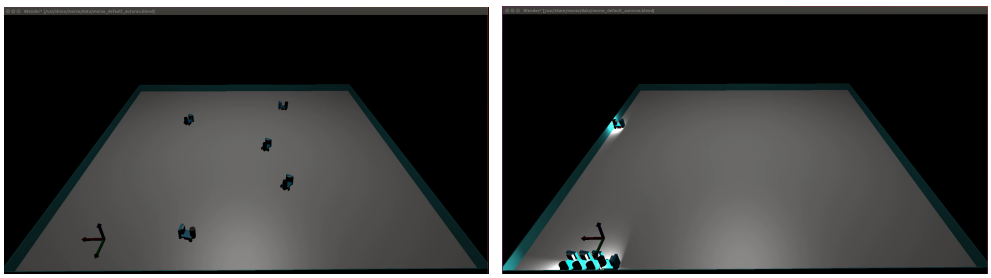
Vídeo del comportamiento:
`https://drive.google.com/open?id=129gaF3GibMCYGMnKKEuG5HVgCTC-D46W`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.49: Agregación en un escenario de superficie cuadrada con una red neuronal entrenada en un escenario con forma de cruz

Vídeo del comportamiento:
`https://drive.google.com/open?id=1HX6jS9_e2iW8QYvaVBuZ7JZfr-PKuTNq`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.50: Agregación en un escenario de superficie cuadrada con una red neuronal entrenada en un escenario con forma de cruz

En conclusión, el resultado del comportamiento varía entre los diferentes entornos, ya que en todos los entornos las colisiones con otros robots impedían que estos pudieran moverse y cambiar de posición. Además, el escenario con forma de cruz ha sido el más complejo para llevar a cabo la tarea, a pesar de haber realizado un entrenamiento en ese escenario.

7.6. Aprendizaje de una conducta de dispersión

En este apartado los robots aprenderán la conducta inversa a la agregación, la dispersión. La función de recompensa será la inversa a la de la agregación. Para el entrenamiento los robots están posicionados cerca, es decir, agrupados, para que la posición inicial no influya negativamente en la recompensa. Además, la red neuronal que se va a entrenar tendrá la misma arquitectura que en el apartado anterior y se va a realizar el entrenamiento con 5 robots.

$$f_{dispersion}(size(l)) = f_{aggregation}(size(l)) \cdot (-1) \tag{7.3}$$

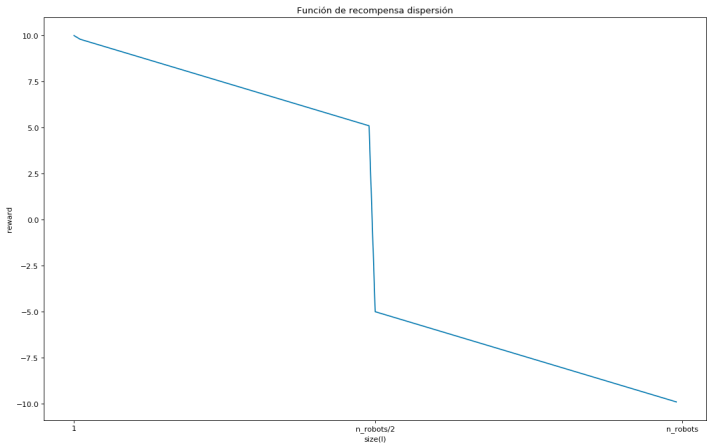


Figura 7.51: Función de recompensa de dispersión

En el entrenamiento de este problema se consiguió la puntuación máxima global desde la

primera generación, aunque, en la mayoría de las generaciones el resultado máximo fuera el máximo global en las evaluaciones del modelo que se hacía cada 25 generaciones obtuvimos el resultado más cercano a la máxima recompensa en la generación 100.

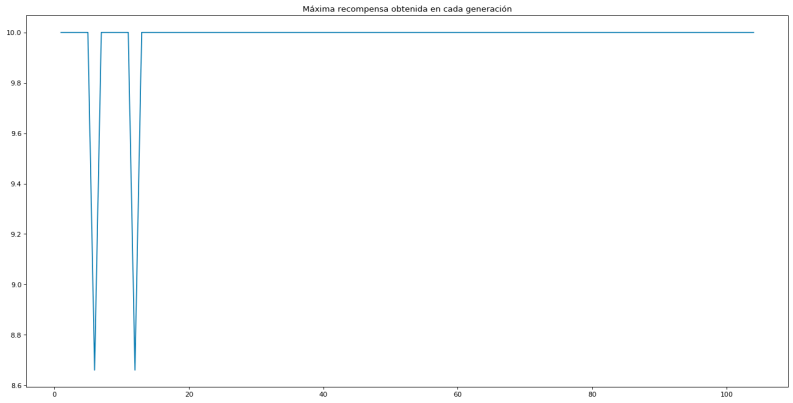


Figura 7.52: Resultados obtenidos en el entrenamiento de dispersión

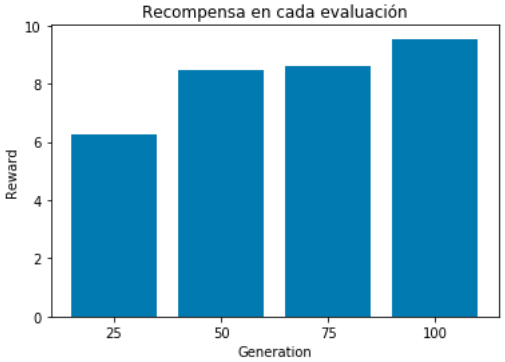


Figura 7.53: Resultados obtenidos en la evaluación durante el entrenamiento de dispersión

El comportamiento de los robots en un entorno 2D cuadrado era encender las luces LED parpadeando mientras se deambulaban por el terreno, a diferencia del comportamiento aprendido de agregación, los robots no se dirigían hacia ningún lugar en concreto.

Vídeo del comportamiento:

`https://drive.google.com/open?id=18EaFf0fvGr4j4rMd2R7nsN-6eJdP_Rix`

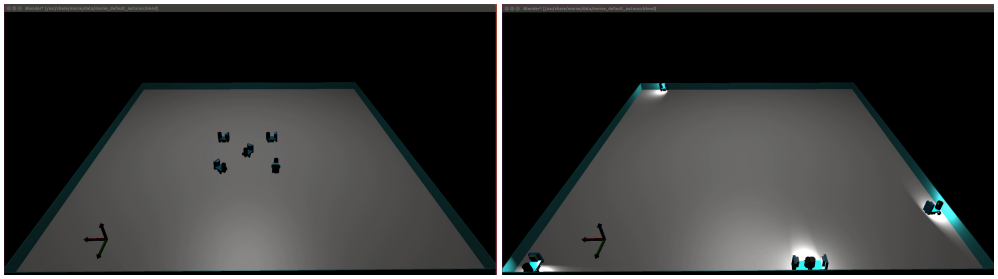
(a) Estado inicial del enjambre **(b)** Estado final del enjambre

Figura 7.54: Dispersión en un escenario de superficie cuadrada en 2D

El comportamiento de los robots en un entorno 3D cuadrado es el mismo que el que tenían en el entorno 2D, aunque algunas veces colisionaban entre ellos y de esta forma perturbaban el comportamiento del otro robot con el que habían colisionado.

Vídeo del comportamiento:

`https://drive.google.com/open?id=1-B-b4_Lx6Y3UJtU9PBjdqPKFd5L3dy4E`



(a) Estado inicial del enjambre **(b)** Estado final del enjambre

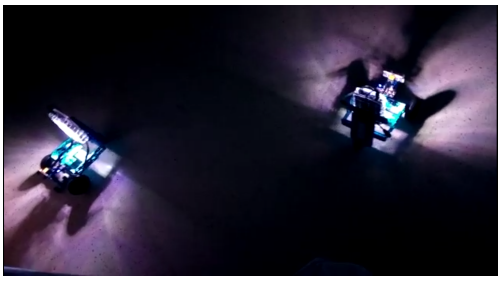
Figura 7.55: Dispersión en un escenario de superficie cuadrada en 3D

Por último, se ha conectado esta red neuronal entrenada con la conducta de dispersión en un entorno real. El comportamiento de los robots en este entorno ha sido el mismo que en el entorno 2D, es decir, estos no se situaban en una localización en concreto ni se reunían.

Vídeo del comportamiento:

`https://drive.google.com/open?id=1nrLKSnhFnTXGicSsP8j639eN0H3nEsU3`

(a) Estado inicial del enjambre



(b) Estado final del enjambre

Figura 7.56: Dispersión en un entorno real

7.7. Conclusión

Al finalizar este capítulo podemos concluir que no es una tarea trivial el aprendizaje de una conducta de agregación, ya que diversos factores como el tamaño del enjambre y el tipo de escenario influyen más en el desempeño de esta tarea que otros factores como la arquitectura de la red neuronal y la estrategia evolutiva seleccionada para el entrenamiento.

En primer lugar, el comportamiento en entornos simulados 2D y 3D frente al comportamiento de los robots reales que se ha observado a lo largo del desarrollo del proyecto es que ha habido diferencias entre los datos obtenidos entre estos tres entornos. Por ejemplo, el sensor ultrasónico del robot real no proporcionaba datos correctos en la lectura de una distancia de un obstáculo si el robot estaba en un ángulo inclinado, ya que, el receptor de este sensor no recibía la señal transmitida. Mientras que en los entornos simulados no estaba este problema porque en el entorno 3D el sensor de ultrasonido fue implementado con un sensor láser, que no presentaba este problema, y en el entorno 2D las distancias de los obstáculos se calculaban mediante el cálculo de distancias de vectores. Otro de los problemas del entorno real es que presentaba cierto ruido en la velocidad de los motores, ya que, el robot tenía problemas a la hora de girar si no se le aplicaba la velocidad adecuada. Además, solía girar más de 90 o en algunos giros porque giraba demasiado rápido para detectar la lectura de la brújula correcta y tenía que dar giros de más.

Por otra parte, ha habido otras diferencias en el resultado de los comportamientos entre los diferentes entornos, aunque, tenían el mismo comportamiento, esto se debe, en parte, a la colisión con otros robots porque perturbaban el comportamiento del otro robot impidiéndole realizar cualquier movimiento. Además, en un entorno simulado si un robot se desestabiliza o se cae al colisionar con una pared, puede volver a su posición original. En cambio, en un entorno real puede que no vuelva a la posición que tenía antes de desestabilizarse. En el desarrollo de este capítulo hemos podido comprobar que en ninguna política se utilizaba el sensor de luminosidad ni las luces LED para la localización del enjambre porque el enjambre se agregaba en una localización en concreto, dándole más importancia a los datos del sensor brújula que a los datos del sensor de luminosidad.

En conclusión, ambos métodos de diseño de comportamientos, el método de diseño basado

en el comportamiento con el algoritmo beeclust y el método de diseño automático con estrategias evolutivas, han logrado su propósito. El comportamiento obtenido con el algoritmo beeclust ha mostrado que es un comportamiento escalable, flexible y robusto, ya que, se ha utilizado en diferentes entornos y en enjambres de diferentes tamaños, 5 robots en el entorno 3D simulado y 3 robots en el entorno real. Sin embargo, este algoritmo no podía asegurar que el enjambre completo se reuniera en una localización porque al deambular por el terreno para explorarlo generaba movimientos aleatorios, lo que provocaba que en algunas ocasiones los robots no podrían encontrar el enjambre. Además, si se perdía el primer robot que se había unido al enjambre antes de que otro robot se uniera a él, el resto de robots no podría encontrar el enjambre porque no podrían detectar el nivel de luz. En cambio, el algoritmo obtenido mediante el método de diseño automático también podía ser escalable, si estaba entrenado con un enjambre de 5 robots, flexible y robusto, ya que, se encontraban siempre en la esquina inferior situada al Oeste. Además, este algoritmo no tenía ningún inconveniente al realizar la tarea si alguno de los robots se perdía. Por lo tanto, sí que se logró una política óptima con el uso de estrategias evolutivas.

8. Conclusiones

Al finalizar este proyecto podemos concluir que se ha conseguido el aprendizaje de una conducta de enjambre con estrategias evolutivas utilizando los robots mBot Ranger. Para el desarrollo de este proyecto se han utilizado dos métodos de diseño distintos para el diseño del comportamiento de enjambre, para el método de diseño basado en el comportamiento se ha utilizado el algoritmo Beeclust, que es un algoritmo basado en el comportamiento de las abejas, y para el método de diseño automático se han utilizado estrategias evolutivas para el aprendizaje por refuerzo.

Por otra parte, se han implementado dos simuladores: el simulador 2D para el desarrollo del proceso de aprendizaje y el simulador 3D para observar el comportamiento obtenido en el aprendizaje antes de conectar la red neuronal a los robots reales. Además, se han analizado y estudiado las diferentes estrategias evolutivas y los diferentes parámetros de la red neuronal para acelerar el proceso de aprendizaje y encontrar una política óptima.

Por último, se han cumplido los objetivos propuestos:

- Se ha diseñado el robot con características específicas para los experimentos, por ejemplo, para la prueba de la plataforma robótica se ha utilizado un diseño distinto que para las tareas de enjambre, ya que, la placa base estaba orientada en distintos ejes según estas tareas. Además, los sensores y actuadores utilizados no eran los mismos.
- Se ha logrado implementar el equilibrio individual de un robot en un balancín.
- Se ha conseguido un equilibrio colectivo utilizando diferentes sistemas de comunicación, aunque, el que ha proporcionado mejores resultados es el equilibrio con sistema centralizado, porque los otros dos tipos de comunicación desestabilizaban el balancín.
- Se ha diseñado de una conducta de agregación con el método de diseño basado en el comportamiento. En este proyecto se ha utilizado el algoritmo Beeclust para desempeñar esta tarea y ha logrado la agregación de todo el enjambre.
- Se ha implementado una simulación realista de los robots mBot Ranger mediante el simulador MORSE, para ello se han sustituido algunos sensores que no existían en esta plataforma, por ejemplo, el sensor ultrasónico se ha sustituido con un sensor láser. Además, se han implementado otros sensores para poder utilizarlos en la simulación como el sensor de luminosidad.
- Para el aprendizaje automático se ha utilizado una simulación rápida. Esta simulación se ha utilizado en el simulador 2D para el uso de la librería Estool.
- Se ha conseguido el aprendizaje de una conducta de agregación. Para esta tarea se ha utilizado la librería de estrategias evolutivas y el simulador 2D. Los resultados que ha ofrecido variaban según el entorno o el número de robots del enjambre utilizados en el entrenamiento, aunque, en general, este método ha proporcionado buenos resultados.

- Se ha desarrollado el aprendizaje de una conducta de dispersión invirtiendo las recompensas de la agregación. Este algoritmo conseguía su propósito, ya que, al final de la ejecución los robots lograban dispersarse.
- Se ha utilizado el diseño de redes neuronales para fomentar el aprendizaje y el funcionamiento del sistema, es decir, se ha comparado el resultado obtenido con diferentes parámetros de la red neuronal utilizada en el proceso de entrenamiento.
- Se han probado varias estrategias evolutivas y se han comparado los resultados que ofrecía cada una en la tarea de agregación. Como esta tarea es una tarea simple, la estrategia que mejores resultados ha proporcionado es SES. Además, con esta estrategia se alcanzaba el máximo global en recompensas.
- La conducta aprendida con simulador simple se ha transferido a simulador complejo, ya que, la red neuronal entrenada con el simulador 2D se ha conectado en el simulador 3D para observar el comportamiento obtenido en una simulación realista.
- La conducta aprendida se ha transferido a los robots reales, en este caso, se ha conectado la red neuronal entrenada en el simulador 2D a los robots reales después de comprobar que el comportamiento obtenido era el correcto utilizando el simulador 3D.
- Se ha realizado un análisis de métrica de las conductas de enjambre como la escalabilidad, robustez y flexibilidad. En los algoritmos obtenidos mediante los dos métodos de diseño, método de diseño basado en el comportamiento y diseño automático, se ha obtenido el mismo resultado en diferentes escenarios y con diferente número de robots del enjambre. Aunque, el escenario de cruz dificultaba esta tarea y en la conducta aprendida mediante estrategias evolutivas impedía que el enjambre se reuniera en un único cluster debido a que la conducta aprendida era que se reunieran en una esquina y en este escenario se reunían en dos esquinas diferentes formando dos grupos.
- Se han realizado pruebas con robots reales. En la prueba de la plataforma robótica no se necesitó el uso de ningún simulador para llevar a cabo esta tarea, por lo que se conectó el algoritmo directamente a los robots reales. Por otra parte, para llevar a cabo la tarea de agregación se conectó primero el algoritmo en los simuladores 2D y 3D y cuando se comprobó que el funcionamiento era el correcto se conectó a los robots reales.

En conclusión, podemos afirmar que se ha obtenido una conducta de enjambre mediante aprendizaje automático con estrategias evolutivas que cumple con los requisitos de la robótica de enjambre.

Bibliografía

Baldassarre, G. (2006). Evolution of collective behaviour: coordination object retrieval in groups of physically-linked simulated robots. Descargado 20/08/2019, de `http://laral.istc.cnr.it/baldassarre/demos/2003swarmobject/swarmobject.htm`

Brambilla, M., Ferrante, E., Birattari, M., y Dorigo, M. (2013). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intell*, *7*, 1–41. doi: 10.1007/s11721-012-0075-2

Campo, A., Nouyan, S., Birattari, M., Groß, R., y Dorigo, M. (2006). Enhancing cooperative transport using negotiation of goal direction. *Lecture notes in computer science: Vol. 4150. Proceedings of the fifth international workshop on ant colony optimization and swarm intelligence (ANTS 2006)*, 365–366.

Christensen, A. L., O'Grady, R., y Dorigo, M. (2009). From fireflies to fault-tolerant swarms of robots. *IEEE Transactions on Evolutionary Computation*, *13*(4), 754–766.

Christensen, A. L., O'Grady, R., y Dorigo, M. (2008). Swarmorph-script: a language for arbitrary morphology generation in self-assembling robots. *Swarm Intelligence*, *2*(2-4), 143–165.

Ducatelle, F., Di Caro, C. P. G. A., y Gambardella, L. M. (2011). Self-organized cooperation between robotic swarms. *Swarm Intelligence*, *5*(2), 73–96.

Garnier, S., Jost, C., Jeanson, R., Gautrais, J., Asadpour, M., Caprari, G., y Theraulaz, G. (2005). Aggregation behaviour as a source of collective decision in a group of cockroach-like robots. *Lecture notes in artificial intelligence, Advances in artificial life*, *3630*, 169–178.

Ha, D. (2017a). Evolving stable strategies. *blog.otoro.net*. Descargado de `http://blog.otoro.net/2017/11/12/evolving-stable-strategies/`

Ha, D. (2017b). A visual guide to evolution strategies. *blog.otoro.net*. Descargado de `http://blog.otoro.net/2017/10/29/visual-evolution-strategies/`

Howard, A., Matarić, M. J., y Sukhatme, G. S. (2002). Mobile sensor network deployment using potential fields: a distributed, scalable solution to the area coverage problem. *Proceedings of the 2002 international symposium on distributed autonomous robotic systems (DARS 2002)*, 299–308.

Ketkar, N. (2017). *Deep learning with python: A hands-on introduction*. Apress. doi: 10.1007/978-1-4842-2766-4

Liu, W. (2007). Modelling of adaptive foraging in swarm robotic systems. Descargado 20/08/2019, de `http://www.brl.ac.uk/researchthemes/swarmrobotics/swarmroboticsystems.aspx`

McLurkin, J., Smith, J., Frankel, J., Sotkowitz, D., Blau, D., y Schmidt, B. (2006). Speaking swarmish: human–robot interface design for large swarms of autonomous mobile robots. *AAAI spring symposium*, 72–75.

Melhuish, C., Holland, O., y Hoddell, S. (1999). Convoying: using chorusing for the formation of travelling groups of minimal agents. *Robotics and Autonomous Systems*, *28*(2-3), 207–216.

Melhuish, C., Welsby, J., y Edwards, C. (1999). Using templates for defensive wall building with autonomous mobile ant-like robots. *Proceedings of towards intelligent and autonomous mobile robots*, *99*.

Mondada, F., Bonani, M., Guignard, A., Magnenat, S., Studer, C., y Floreano, D. (2005). Superlinear physical performances in a swarm-bot. , *In Lecture notes in computer science: Vol. 3630. Proceedings of the VIIIth European conference on artificial life*, 282–291.

Montes de Oca, M. A., Ferrante, E., Scheidler, A., Pinciroli, C., Birattari, M., y Dorigo, M. (2011). Majority-rule opinion dynamics with differential latency: a mechanism for self-organized collective decision-making. *Swarm Intelligence*, *5*(3-4), 305–327.

Nouyan, S., Groß, R., Bonani, M., Mondada, F., y Dorigo, M. (2009). Teamwork in self-organized robot colonies. *IEEE Transactions on Evolutionary Computation*, *13*(4), 695–711.

Pini, G. (2011). Task partitioning in swarms of robots an adaptive method for strategy selection. Descargado 20/08/2019, de `http://iridia.ulb.ac.be/supp/IridiaSupp2011-003/index.html`

Reynolds, C. W. (1987). Flocks, herds and schools: a distributed behavioral model. *Computer Graphics*, *21*(4), 25–34.

Schmickl, T., Hamann, H., y Karl-Franzens. (2016). Beeclust: A swarm algorithm derived from honeybees. derivation of the algorithm, analysis by mathematical models and implementation on a robot swarm. , 1–45.

SOYSAL, O., BAHÇECİ, E., y ŞAHİN, E. (2007). Aggregation in swarm robotic systems: Evolution and probabilistic control. *TURKISH JOURNAL OF ELECTRICAL ENGINEERING & COMPUTER SCIENCES*, *15*(2), 199–225.

Spears, W. M., y Spears, D. F. (2012). Physics-based swarm intelligence.

Sperati, V., Trianni, V., y Nolfi, S. (2011). Self-organised path formation in a swarm of robots. *Swarm Intelligence*, *5*, 97–119.

Trianni, V., Groß, R., Labella, T. H., Şahin, E., y Dorigo, M. (2003a). Evolving aggregation behaviors in a swarm of robots. *ECAL 2003, LNAI 2801*, 865–874.

Trianni, V., Groß, R., Labella, T. H., Şahin, E., y Dorigo, M. (2003b). Evolving aggregation behaviors in a swarm of robots. *Lecture notes in artificial intelligence. Advances in artificial life: 7th European conference—ECAL*, *2801*(4), 865–874.

TShucker, B., y Bennett, J. K. (2007). Scalable control of distributed robotic macrosensors. *Distributed autonomous robotic systems*, *6*, 379–388.

Turgut, A. E., Çelikkanat, H., Gökçe, F., y Şahin, E. (2008). Self-organized flocking in mobile robot swarms. *Swarm Intelligence*, *2*(2-4), 97–120.

Werfel, J. (2011). Distributed multi-robot algorithms for the termes 3d collective construction system. Descargado 20/08/2019, de `http://www.eecs.harvard.edu/ssr/publications/`

Lista de Acrónimos y Abreviaturas

CMA-ES	Estrategia evolutiva de adaptación de la matriz de covarianza.
GA	Algoritmos genéticos simples.
NES	Estrategias evolutivas naturales.
OpenES	OpenAI Estrategia evolutiva.
PEPG	Gradientes de política de exploración de parámetros.
PFSMs	Diseño probabilístico de máquinas de estados finitos.
SES	Estrategia evolutiva simple.

A. Anexo I

En este anexo se va a explicar el proceso de instalación de la librería AurigaPy y la conexión de los robots mBot Ranger mediante Bluetooth en sistemas Linux.

A.1. Instalación de la librería AurigaPy

Para poder utilizar la librería en Linux hay que seguir los siguientes pasos:

1. Abrir el archivo: .bashrc
2. Añadir al final del archivo la línea para modificar la variable PYTHONPATH:

Código A.1: Modificar variable PYTHONPATH

```
1        export PYTHONPATH=directorio_libreria
```

3. Reiniciar el terminal o reiniciar su configuración con el comando:

Código A.2: Reiniciar configuración del terminal

```
1        source ~/.bashrc
```

Cuando ya hemos configurado el paquete podemos ejecutar en el robot cualquier programa en Python si conocemos el puerto serie de bluetooth o de USB.

A.2. Conexión mediante Bluetooth

Cuando ya hemos configurado el paquete podemos ejecutar en el robot cualquier programa en Python si conocemos el puerto serie de bluetooth o de USB. Como comenté anteriormente, tenemos un inconveniente a la hora de conocer el puerto serie del Bluetooth al que se conecta el robot porque en la última versión de mBlock para Linux no se puede saber desde qué puerto serie se está conectando.

Este problema se soluciona introduciendo los siguientes comandos una vez encendido el robot:

1. Código A.3: Escanear los dispositivos bluetooth

```
1        hcitool scan
```

Código A.4: Asociar el puerto serie a la MAC del robot

```
1        sudo rfcomm bind /dev/nombreRobot mac_robot
```

3. Código A.5: Confirmar conexión

```
1        rfcomm show /dev/nombreRobot
```

Código A.6: Permitir que nuestro usuario tenga acceso al puerto serie

```
1        sudo chown usuario /dev/nombreRobot
```

Antes de seguir estos pasos debemos tener el bluetooth activado y debemos cerrar el programa mBlock. Los pasos 2 y 4 se tienen que hacer cada vez que reiniciamos el equipo.

B. Anexo II

En este anexo se va a explicar el entrenamiento de una red neuronal mediante estrategias evolutivas con la librería Estool y la comparación de resultados con el test de Wilcoxon.

B.1. Librería estool

La librería Estool, Ha (2017a), dispone de varios problemas resueltos de OpenAI. Además, permite crear nuevos entornos e incluirlos para poder obtener una solución mediante estrategias evolutivas.

Para encontrar la solución a un problema utilizando esta librería con una población de 32 individuos y evaluando cada política 3 veces se puede utilizar el siguiente comando, donde ***es*** es la estrategia evolutiva a utilizar.

Código B.1: Ejecutar entrenamiento librería Estool

```
1   python train.py name_of_environment -e 3 -n 8 -t 4 -o es
```

Para el desarrollo de este proyecto se implementó un entorno con Box2D y PyGame y se incluyó en la librería para poder obtener la óptima política.

Esta librería almacena el resultado de cada generación, así como la mejor red neuronal encontrada cada 25 generaciones. Algunos de los resultados que almacena son: El peor resultado, el mejor resultado, la media de los resultados de todas las poblaciones o el tiempo que tarda en ejecutar una generación. Estos resultados se almacenan en un fichero Json que se puede utilizar para obtener los datos del entrenamiento.

La red neuronal resultante del entrenamiento también se almacena en un fichero Json y se puede cargar con el siguiente comando.

Código B.2: Ejecutar modelo entrenado librería Estool

```
1   python model.py name_of_environment log/name_of_your_json_file.best.json
```

B.2. Test de Wilcoxon

El test de Wilcoxon o la prueba de los rangos con signo de Wilcoxon, es una prueba no paramétrica que determina si dos muestras son iguales o no. Existe una librería en Python que implementa este test y puede indicar si una muestra es mayor o menor que otra. El ***pvalue*** devuelto por la función ***wilcoxon*** en Python es el resultado de ambas colas de la distribución, si queremos el resultado de una de ellas se debe dividir este entre dos, es decir, utilizar ***pvalue/2*** en vez de ***pvalue***.

Código B.3: Ejemplo test de Wilcoxon en Python

```
from scipy.stats import wilcoxon
statistic, pvalue = wilcoxon(x, y, zero_method='wilcox',
                        alternative = "greater")
```

Si ***pvalue/2*** es menor o igual que α, nivel de significancia estadística, se rechaza la hipótesis nula, H_0, esta indica que dos o más muestras no tienen relación entre si, por lo tanto, sí se puede afirmar que $\boldsymbol{x}$ es mayor que $\boldsymbol{y}$. En cambio, si obtenemos que ***pvalue/2*** es mayor que α no se puede rechazar H_0, por lo que no podemos asegurar que ambas muestras sean diferentes. Se establece que el nivel de significancia estadística, α, es 0.05, ya que este valor indica un riesgo del 5% de concluir que exista una diferencia cuando no hay diferencia real. En resumen, si al aplicar el test de Wilcoxon se obtiene un valor de ***pvalue/2*** menor o igual que 0.05 se puede afirmar que $\boldsymbol{x}$ es mayor que $\boldsymbol{y}$ con una confianza del 95%.